MIT/LCS/TR-276

AUTOMATIC SYNTHESIS OF IMPLEMENTATIONS FOR ABSTRACT DATA TYPES FROM ALGEBRAIC SPECIFICATIONS

Mandayam K. Srivas

This blank page was inserted to preserve pagination.

Automatic Synthesis of Implementations for Abstract Data Types from Algebraic Specifications

by

Mandayam Kannappan Srivas

June 1982

This research was supported (in part) by the National Science Foundation under grant MCS78-01798 and by the Defense Advanced Research Projects Agency monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge **MA 02139**

Automatic Synthesis of Implementations for Abstract Data Types from Algebraic Specifications

Abstract

Algebraic specifications have been used extensively to prove properties of abstract data types and to establish the correctness of implementations of data types. This thesis explores an automatic method of synthesizing implementations for data types from their algebraic specifications.

The inputs to the synthesis procedure consist of a specification for the implemented type, a specification for each of the implementing types, and a formal description of the representation scheme to be used by the implementation. The output of the procedure consists of an implementation for each of the operations of the implemented type in a simple applicative language.

The inputs and the output of the synthesis procedure are precisely characterized. A formal basis for the method employed by the procedure is developed. The method is based on the principle of reversing the technique of proving the correctness of an implementation of a data type. The restrictions on the inputs, and the conditions under wliich the procedure synthesizes an implementation successfully are formally characterized.

Name and Title of Thesis Supervisor: John V. Guttag
Associate Professor of Computer Science and Engineering

Key Words and Phrases: Abstract Data Type, Algebraic Specification, Association Specification, Abstraction Function, Invariant, Preliminary Implementation, Target Implementation, Term Rewriting System, Principle of Definition, Reduction, Expansion.

---

1. This report is a minor revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science in December 1981 in partial fulfillment of the requirements for the degree of Doctor Philosophy.

Acknowledgments

I want to express my appreciation to my thesis supervisor, John Guttag, for all the help he has given me. His patience, encouragement, support, and constructive criticisms of the work at various stages of its development have been invaluable.

Each of my thesis readers, Arvind, Barbara Liskov, and Dave Musser, has contributed important insights to different aspects of both the research and the presentation of the thesis. I am especially thankful to Barbara for suggesting to me the topic of the thesis, and to Dave for his careful reading of several drafts of the thesis which made the final version substantially more readable.

My officemates, Deepak Kapaur, Pierre Lescanne, and Carl Seaquist, have helped me in many ways during the research. They gave me a patient audience whenever I needed, and read drafts of the thesis. Deepak was especially helpful in organizing my ideas at the initial stages of the research.

Many people at M.I.T., graduate students in the Laboratory of Computer Science and outside, have helped create an interesting, stimulating, often diverting, but always supportive environment in which to work. I want to thank in particular Betty, Jeannette, Sriram, Ravi, and Kanchan for their continual encouragement.

This research was supported (in part) by the National Science Foundation under grant MCS78-01798 and by the Defense Advanced Research Projects Agency monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

CONTENTS

FIGURES

1. Introduction

1.1 Goals of the Thesis

This thesis is concerned with the problem of automatic synthesis of implementations for abstract data types from their algebraic specifications. The inputs to the synthesis procedure include (i) a formal specification of the data type to be implemented, (ii) a formal specification of each of the implementing types, and (iii) a formal description of the representation scheme to be used by the desired implementation. The output consists of an implementation for each of the operations of the implemented type. The inputs are specified using an algebraic specification technique [14, 18, 25].

The thesis has three main goals:

(1) To precisely characterize both the inputs of the synthesis procedure, and the output.

(2) To devise an automatic method of deriving the output from the inputs.

(3) To provide a formal basis for the method.

The method of derivation is described in terms of a set of *synthesis rules*. The output is derived by invoking the synthesis rules a finite number of times. The thesis describes how the synthesis rules are used in deriving a suitable implementation.

The purpose of providing a formal basis for the method is to justify the correctness of the implementations derived by the synthesis procedure. The formal basis also helps in characterizing the scope of the synthesis procedure.

1.2 Motivation for The Research

The reliability of computer software has received a great deal of attention in recent years. Rapid advances in hardware technology have dramatically decreased the cost of hardware relative to software. As a result, the cost of producing and maintaining software has become a major concern. An effective way of improving the reliability and the cost of software simultaneously is to find methods to decrease the effort required to produce correct software. At present, active research is underway [43] in exploring this avenue. Several

approaches have been proposed, each of which can be put under one of the following three categories based on the degree of automation it offers: manual approaches, semi-automatic approaches, and automatic approaches.

The manual approach advocates discipline in human programming [31, 11, 41]. It consists of identifying new mechanisms of abstractions [32] that encourage the advocated discipline. The most significant contribution of this approach has been the inducement of a change in the attitude of programmers towards the style of programming. Concrete manifestations of this change include the birth of the concept of abstract data types, and the development of new languages [34, 29, 52] to support data types.

The goal of the semi-automatic approach is to seek machine help to establish the correctness of programs written by the user. Formal methods are developed to specify and verify properties of pieces of software [13, 12, 20]; systems are built to carry out verification automatically or semi-automatically [27, 15]. A variant of the verification method is the programmer's apprentice method [19]. The programmer's apprentice provides an interactive programming environment built up by a set of tools which helps the programmer in preparing and checking his work in several ways. The tools range from simple editors to more sophisticated ones that can analyze and criticize a user's program during the various phases of programming. Yet another way of providing partial machine help is to build systems [2, 3, 48] that will help apply transformation rules chosen from a catalogue of equivalence preserving transformations. The programmer can refine or improve the efficiency of his programs by judiciously choosing the appropriate rules from the catalogue.

The automatic approach, under which our research falls, seeks to automate a part or all of the programming process itself. Its goal is to generate code for programs from their high-level declarative descriptions, thereby relieving the programmer of having to worry about error-prone, low-level details of programming. Though this may one day be feasible, experience [1, 36] in the last few years shows that not nearly enough is known about the process to automate it completely. Two remedies have been used with some success to break the stalemate in the situation: The first is to restrict the domain for which programs are being synthesized [4]; the second is to expect the user to furnish more information about the desired properties of the program [6] to guide the synthesis procedure.

A third course of action that has not so far been employed in earnest is to complement the automatic approach with recent advances in programming methodology. (Bauer, et.al., [3] have employed this idea with the semi-automatic approach.) In particular, the idea of designing software as a hierarchy of abstractions can be used to aid the synthesis procedure. Such a hierarchical design for the program reduces the amount of refinement required to be performed by the synthesizer at each step.

The thesis takes into consideration all the factors mentioned above. Within the general area of programming, we restrict ourselves to the study of synthesis of implementations for abstract data types. We believe that the synthesis of implementations for abstract data types is amenable to automation because the specification techniques for data types have been extensively studied, and hence, are better understood. We also expect additional information about the implementation to be furnished by the user. This information is provided in the form of a description of the representation scheme to be used by the implementation.

1.3 Related Work

The works related to ours lie partly in the area of general program synthesis and partly in the area of automatic implementation of data structures.

In the general area of synthesis, the work most closely related to ours is that of Darlington [8, 9]. He has developed a system that uses a set of transformation rules to improve semi-automatically the efficiency of recursive programs and also to construct new recursive programs. Recently, he has also applied the transformation rules to synthesize implementations for data types [7]. The synthesis rules developed in the thesis are closely related to his. The difference lies in the method in which the synthesis rules are used to synthesize implementations. Our method is based on verification techniques of data types. Our work has two advantages over his. Firstly, the class of implementations derived by our method is larger than his. This is because we develop more ways of using the synthesis rules for deriving implementations. Secondly, we formally characterize the conditions under which the synthesis rules yield a correct implementation for data types.

The ZAP system [30] of Feather's is a program transformation system in which the basic rules of manipulations are similar to our synthesis rules. His work is different from ours in two ways. Firstly, he is concerned with developing higher level stretegies to apply the basic tranformation rules (in general, any equivalence preserving rules) for the construction of large-sized programs. Secondly, his approach is less automatic than ours. The emphasis in the design of ZAP is to use "metaprograms" to improve communication between the user and the system. There are two inputs to ZAP: the specification of the program to be constructed and a metaprogram which consists of a sequence of commands that direct the transformation process. The metaprogram expresses the higher level strategy to be used in applying the tranformation rules.

Within the area of automatic implementations for data structures, the work of Okrent [40] has goals closest to ours. Okrent's method uses only the algebraic specifications of the data types involved as inputs. Because of the lack of information about the desired representation scheme, the implementations generated by his synthesis procedure are not as interesting as the ones generated by ours. He limits severely the range of the data types acceptable as inputs. He also concentrates on a fixed set of target structures such as contiguous memory and heap memory for the implementations.

Another work in this area that is related to ours is that of Subrahmanyam's [50]. Subrahmanyam's method like Okrent's does not use any information about the representation scheme. His method has a provision for the user to specify performance constraints on the desired implementation. The method is based on partitioning the operation set of the data type into a kernel set and a nonkernel set. Implementations for the kernel operations are derived by identifying pairs of functions (on the representation type) called *retrievable insertion function pairs.* Implementations for the nonkernel operations are derived in terms of the implementations for the kernel operations so as to meet the performance constraints.

Most of the other research in the automatic generation of data structure implementations has been concerned with the automatic selection of an optimal representation for data structures. Rowe and Tonge [47], Rovner [46], and Tompa and Gotlieb [51] have studied optimization problems for a language containing a fixed set of high

level data structures. First they build a library of possible implementations for each fixed high level data structure in the language, along with a parameterized description of the performance of each library entry. Then they proceed to select the "best" implementation for each instance of the data structure, by making a flow analysis of the program that uses the data structure. The goal of our work is to derive an implementation for a given representation rather than to select an optimal one among a given set of representations.

Standish, et.al., [49], Bauer, et.al., [3], and Wile, et.al, [2] have developed catalogues of equivalence preserving transformation rules as a part of program development systems. The programmer can refine or improve the efficiency of his programs by instructing the system to apply appropriate transformation rules on the programs. None of these works, however, deals explicitly with the implementation of data types. It is possible, with some modifications, to incorporate our synthesis rules as a part of their system.

1.4 Organization of the Thesis

The next chapter gives an overview of the synthesis procedure. The third chapter describes in detail the inputs of the synthesis procedure, and formalizes the restrictions on the inputs. The synthesis procedure derives an implementation in two stages: The implementation is first derived in a preliminary form which is then transformed into a final form. The first stage of the procedure is the topic of the fourth and the fifth chapters. The sixth chapter describes the second stage. The last chapter gives the concluding remarks.

2. An Overview of the Synthesis Procedure

This chapter gives an overview of the synthesis procedure. The first section gives a scenario of the synthesis procedure from a user's point of view. It briefly describes the form of the inputs to the synthesis procedure, and the form of its outputs via an example. The second section gives a summary of the synthesis procedure. It points out the nontrivial issues involved in the method employed by the procedure for deriving an implementation. The last section describes the scope of the procedure.

2.1 The User's View

Consider the following scenario involving a programmer. The programmer has designed an abstract data type (the *implemented type*) to be used in solving one of his programming problems. He is now seeking the help of a system for implementing the type using another data type, called the *representation type*; The representation type is chosen by the user himself. Furthermore, he is willing to furnish information about how he wants the values of the representation type to be used in representing the values of the implemented type. The system is expected to generate automatically (or with some help from the user) an implementation for the implemented type that uses the representation type as the representation in a manner consistent with that suggested by the user.

Viewed as a black box, the inputs to the procedure are:

(i) A specification of the implemented type,

(ii) a specification of the representation type, and specifications of all the types used in the specification of the representation type. We refer to the representation type, and all the types its specification uses as the *implementing types.*

(iii) an *association specification* that describes how the values of the representation type are to be used in representing the values of the implemented type; this corresponds to the *representation (or abstraction) function* defined by Hoare in [21].

The output of the synthesis procedure consists of an implementation for each of the

operations of the implemented type in terms of the operations of the implementing types. To get a better idea about the inputs and the output, let us consider an example of deriving an implementation for the data type **Queue_Int** in terms of **Circ_List**. **Queue_Int** is a first-in-first-out queue of integers. Elements are added to a queue at the rear end, and removed from the front end. **Circ_List** is a list of integers. Elements are inserted into and removed from a list at the same end, which is the rear end of the list. The operation that gives **Circ_List** a circular character is **Rotate**. **Rotate** moves every element in a list by one position towards the rear end in a cyclic fashion, i.e., the element at the rear end is moved to the front end.

In this example, the implemented type is **Queue_Int** and the representation type is **Circ_List**. **Circ_List** uses (this notion is defined precisely in the next chapter) the data types **Integer** and **Bool**, so the implementing types include **Circ_List**, **Integer**, and **Bool**. Figures 1, 2, and 3 give the inputs to the synthesis procedure. (The figures also give an informal description of the operations of the data types.) Specifications of **Integer** and **Bool** should also be given as inputs, although we have not shown them here. The language used to express the data type specifications is equational, similar to the ones developed in [14, 18, 25]. One of the crucial differences is the following: We assume that the specification of every data type identifies a *basis* for the data type. A basis is a minimal set of operations of the data type that can be used to generate all the values of the type. The operations in the basis are called the *generators* of the type. For example, the operations **Create** and **Insert** can be the generators for **Circ_List**. The specification language is described in the next chapter.

Fig. 3 gives the association specification for the implementation to be derived. It characterizes the representation scheme to be used by the implementation. The association specification is expressed in two parts. The first part specifies the *invariant* $\mathfrak{I}$. $\mathfrak{I}$ is a predicate that specifies the set of values that may be used to represent the values of the implemented type; only those values of the representation type for which $\mathfrak{I}$ is **True** may be used to represent the values of the implemented type. In the present example, $\mathfrak{I}$ is **True** for all values of **Circ_List**. The second part specifies the *abstraction function* $\mathcal{A}$; $\mathcal{A}$ maps a value the representation type to the value of the implemented type that the former may represent. In the present example $\mathcal{A}$ specifies the following mapping: The empty queue is represented by

Fig. 1. Specification of Queue_Int

Queue_Int is Nullq, Enqueue, Front, Dequeue, Append, Size

Defining Types

Bool, Int

Operations

Nullq : -> Queue_Int
Enqueue : Queue_Int X Int -> Queue_Int
Front : Queue_Int -> Int ∪ { ERROR }
Dequeue : Queue_Int -> Queue_Int ∪ { ERROR }
Append : Queue_Int X Queue_Int -> Queue_Int
Size : Queue_Int -> Int

Comment

Queue_Int is a FIFO queue of integers. **Nullq** constructs the empty queue. **Enqueue** adds an element to a queue at the rear end. **Dequeue** removes the element at the front of a queue. **Front** returns the element at the front of a queue. **Append** joins two queues adding the elements of the second argument at the rear of the first argument. **Size** computes the number of elements in a queue.

Basis

{ Nullq, Enqueue }

Axioms

(1) Front(Nullq) ≡ ERROR
(2) Front(Enqueue(Nullq, e)) ≡ e
(3) Front(Enqueue(Enqueue(q, e1), e2)) ≡ Front(Enqueue(q, e1))

(4) Dequeue(Nullq) ≡ ERROR
(5) Dequeue(Enqueue(Nullq, e)) ≡ Nullq
(6) Dequeue(Enqueue(Enqueue(q, e1), e2)) ≡ Enqueue(Dequeue(Enqueue(q, e1)), e2)

(10) Append(q, Nullq) ≡ q
(11) Append(q1, Enqueue(q2, e2)) ≡ Enqueue(Append(q1, q2), e2)

(12) Size(Nullq) ≡ 0
(13) Size(Enqueue(q, e)) ≡ Size(q) + 1

Fig. 2. Specification of Circ_List

Circ_List is Create, Insert, Value, Remove, Rotate, Empty, Join

Defining Types

Integer, Boolean

Operations

Create	**: -> Circ_List**
Insert	**: Circ_List X Integer -> Circ_List**
Value	**: Circ_List -> Integer ∪ { ERROR }**
Remove	**: Circ_List -> Circ_List ∪ { ERROR }**
Rotate	**: Circ_List -> Circ_List**
Empty	**: Circ_List -> Boolean**
Join	**: Circ_list X Circ_list -> Circ_list**

Comment

Circ_List is a list of integers with a front end and a rear end. **Create** constructs an empty list; the front and the rear ends of an empty list are the same. **Insert** inserts an element into a list at the rear end. **Value** returns the element at the rear end of a list. **Remove** removes the element at the rear end from a list. **Rotate** moves every element in a list by one position towards the rear end in a cyclic fashion, i.e., the element at the rear is moved to the front. **Empty** checks if a list is empty. **Join** joins two lists by positioning the first argument in front of the second.

Basis

{Create, Insert}

Axioms

(1) Value(Create) ≡ ERROR
(2) Value(Insert(c, i)) ≡ i

(3) Remove(Create) ≡ ERROR
(4) Remove(Insert(c, i)) ≡ c

(5) Rotate(Create) ≡ Create
(6) Rotate(Insert(Create, i)) ≡ Insert(Create, i)
(7) Rotate(Insert(Insert(c, i1), i2))) ≡ Insert(Rotate(Insert(c, i2)), i1)

(8) Empty(Create) ≡ true
(9) Empty(Insert(c, i)) ≡ false

(10) Join(c, Create) ≡ c
(11) Join(c, Insert(d, i)) ≡ Insert(Join(c, d), i)

Fig. 3. Association Specification

Invariant

$\mathfrak{I}$(c) ≡ True

Abstraction Function

$\mathcal{A}$(Create) ≡ Nullq
$\mathcal{A}$(Insert(c, i)) ≡ add_at_head($\mathcal{A}$(c), i)

add_at_head(Nullq, i) ≡ Enqueue(Nullq, i)
add_at_head(Enqueue(q, i), i1) ≡ Enqueue(add_at_head(q, i1), i)

the empty list. A nonempty queue is represented by a list whose elements are identical to the ones in the queue, but are arranged in the reverse order. The motivation for this representation scheme is that reading and deletion of elements from a queue can be performed efficiently. Note that the specification of $\mathcal{A}$ uses an auxiliary function **Add_at_head** on **Queue_Int**; the auxiliary function adds an element at the front end of a queue.

Fig. 4 shows the output of the synthesis procedure. The output defines a set of functions, called the *implementing functions*, on **Circ_List**. Every implementing function implements an operation of **Queue_Int**. The implementing function implementing the operation **f** is given the name **F**. For instance, **NULLQ** implements **Nullq**. The target

Fig. 4. An Implementation

```
NULLQ() :: = Create()

ENQUEUE(c, j) :: = Rotate(Insert(c, j))

FRONT(c) :: = Value(c)

DEQUEUE(c) :: = Remove(c)

APPEND(c, d) :: = Join(d, c)

SIZE(c) :: = if Empty(c) then 0
             else SIZE(Remove(c)) + 1
```

language used to express the implementations for the operations is a simple applicative language. The only mechanisms available in the language to build programs are: functional composition, conditional expressions, and recursive function definition. The language uses a method of defining function that is customarily used in applicative languages like pure LISP [37]. A function F is defined using the following schema: $F(v_1, \ldots, v_k) ::= e$, where $v_1, \ldots, v_k$ are variables, and e is an expression containing those variables. A function definition may use the operations of the implementing types as base functions.

2.2 A Summary of the Synthesis Procedure

The synthesis procedure is summarized in an illustrative fashion using the example already introduced. This is done in the first two subsections. In the example introduced, the invariant $\mathfrak{I}$ is a trivial one: It is **True** on all values. In the third subsection, we highlight the issues involved in deriving an implementation in the presence of a nontrivial invariant by introducing a new example.

The method used by the procedure to derive an implementation is based on treating every equation in the specifications as a *rewrite rule*.[2] The procedure begins by combining all the input specifications into a rewriting system called the *Initial World* (IW). (IW is obtained by simply replacing the symbol $\equiv$ by $\rightarrow$ in the input specifications.) The procedure assumes that IW satisfies the *uniform termination property* as well as the *unique termination property*. (IW is said to be *convergent* in such a case. This is similar to the Church-Rosser property.) The uniform termination property ensures that every chain of reductions starting from an expression terminates. The unique termination property ensures that all chains of reductions starting from an expression terminate in the same expression. These two properties ensure that the equivalence relation characterized by a specification can be computed by using the rules in IW for reducing expressions. The procedure also assumes that there is a predefined

---

2. A rewrite rule (written $\alpha \rightarrow \beta$) is an ordered pair - a left hand side and a right hand side - of expressions. A rewrite rule can be used to *reduce* any expression that is an instance of the left hand side into an expression that is an instance of the right hand side. A *rewriting system* is a set of rewrite rules.

termination ordering ($\succ$) on expressions which can be used for showing the uniform termination property of rewriting systems.

The synthesis procedure derives the implementation in two stages. In the first stage the procedure derives the implementation in an intermediate form. The intermediate form is called a *preliminary implementation.* In the second stage the preliminary implementation is transformed into an implementation in the target language (*target implementation*). Fig. 5 gives a preliminary implementation for **Queue_Int** that is consistent with the association specification given in Fig. 3. There are two crucial differences between a preliminary implementation and a target implementation. The first one concerns the methods used for defining the implementing functions. A preliminary implementation defines a function as a set of rewrite rules. The rewrite rules defining an implementing function **F** are the ones that have **F** as the outermost symbol on their left hand side. For instance, rules (2) and (3) in Fig. 5 define **ENQUEUE**. The second difference is that the only operations of the representation type that are permitted to appear in a preliminary implementation are its generators. A target implementation is permitted to use all the operations of the representation type. In the example under consideration, for instance, a preliminary implementation may use all the operations of **Integer** and **Bool**, but only the generators

---

Fig. 5. A Preliminary Implementation

(1) NULLQ() → Create()

(2) ENQUEUE(Create, j) → Insert(Create, j)
(3) ENQUEUE(Insert(c, i), j) → Insert(ENQUEUE(c, j), i)

(4) FRONT(Create) → ERROR
(5) FRONT(Insert(c, i)) → i

(6) DEQUEUE(Create) → ERROR
(7) DEQUEUE(Insert(c,i)) → c

(8) APPEND(c, Create) → c
(9) APPEND(c, Insert(d, i)) → APPEND(ENQUEUE(c, i), d)

(10) SIZE(Create) → 0
(11) SIZE(Insert(c, i)) → SIZE(c) + 1

(Create, and Insert) of Circ_List.

There are two reasons for the decomposition. Firstly, it makes the synthesis procedure more modular. Target language dependent transformations are separated from the language independent transformations. The decomposition also lends itself naturally to deferring efficiency improving transformations to the later stage. In the first stage one can concentrate on deriving a simple correct implementation. Secondly, the decomposition reduces the complexity of the structure of synthesis procedure. The first stage deals with the techniques for deriving an implementation from the specification of the data type. The second stage deals with the techniques for deriving alternate forms of implementations from an preliminary implementation. The decomposition provides a better insight into the synthesis method, and simplifies the description of the synthesis procedure. The next two subsections give an overview of the two stages of the synthesis procedure.

2.2.1 Stage 1: Preliminary Implementation Derivation

A preliminary implementation of a data type is correct with respect to an abstract function $\mathcal{A}$ if the following condition holds: Every implementing function **F** (that implements the operation **f**) defined by the preliminary implementation is a total function on the representation values so that the homomorphism property $\mathcal{H}(F(x)) = f(\mathcal{H}(x))$ holds. Here $\mathcal{H}$ is a function on the values of the implementing types; $\mathcal{H}$ behaves exactly like the abstraction function $\mathcal{A}$ on the representation values, and like an identity function on all other values. The synthesis procedure derives a preliminary implementation so that the above criterion of correctness is satisfied.

The procedure synthesizes the preliminary implementation for one operation at a time by deriving a separate set of rewrite rules for every operation. Since the method used is the same for every operation, we illustrate the synthesis of only a couple of operations. The procedure first determines the left hand sides of all the rules of the preliminary implementation. Then, it determines a suitable right hand side for each of the rules.

2.2.1.1 Determining the Left Hand Side

One of the correctness requirements of a preliminary implementation is that it must define a total function on the representation type. This requirement is ensured by deriving the rules of the preliminary implementation so that (1) they satisfy the uniform termination property, and (2) they are *well-spanned.* The first property is ensured while deriving the right hand side of the rules. The second property is used to determine the left hand sides.

The second property requires the left hand side expressions of the rules to be of a particular form. For instance, any pair of rules that have the form given below constitute a well-spanned set of rules for **ENQUEUE**. (In the following $?rhs_1$ and $?rhs_2$ are used as place holders for expressions to be determined later.)

ENQUEUE(Create, j) $\rightarrow ?rhs_1$

ENQUEUE(Insert(c, i), j) $\rightarrow ?rhs_2$

Note that the left hand side of each of the above rules consists of **ENQUEUE** applied to arguments that are generator expressions.[3] The set of arguments, i.e., sequences of generator expressions, to **ENQUEUE** on the left hand side of the rules is **ArgsSet** = {<**Create, j**>, <**Insert(c, i), j**>}. **ArgsSet** spans the set of all ordered pairs of generator constants. In other words, every pair of generator constants is an instance of one of the arguments in **ArgsSet**. This property ensures that the definition of **ENQUEUE** accounts for all the representation values. It is easy to build a procedure that automatically generates a well-spanned **ArgsSet**, once the generators of the representation type are identified. Thus, an appropriate set of left hand sides for the rewrite rules to be derived can be determined automatically.

---

3. A generator expression is an expression in which the only function symbols involved are the generators. A generator constant is a generator expression that does not contain any variables.

2.2.1.2 Determining the Right Hand Side

The right hand sides of the rules are determined so that the preliminary implementation satisfies the homomorphism property mentioned earlier. For this, the Initial World, IW, is first supplemented with a set of rules, called the $\mathcal{H}$-rules. The $\mathcal{H}$-rules express the homomorphism property; there is an $\mathcal{H}$_rule for every implementing function. For instance, the $\mathcal{H}$-rule corresponding to **ENQUEUE** is $\mathcal{H}$(**ENQUEUE(c, j)**) → **Enqueue**($\mathcal{H}$(**c**), $\mathcal{H}$(**j**)). Let us call the supplemented system the *Perturbed World* (**PW**).[4]

The Perturbed World (**PW**) is then used to derive a set of *synthesis equations*, one equation for every rule in the preliminary implementation. The right hand side of a rule is determined from the right hand side of the corresponding synthesis equation. For instance, the synthesis equation corresponding to the rule **ENQUEUE(Insert(c, i), j)** → **?rhs**$_2$ is an equation of the form $\mathcal{H}$(**ENQUEUE(Insert(c, i), j)**) ≡ $\mathcal{H}$(**?rhs**$_2$) that satisfies the following conditions:

(1) $\mathcal{H}$(**ENQUEUE(Insert(c, i), j)**) ≡ $\mathcal{H}$(**?rhs**$_2$) is a theorem of **PW**

(2) **ENQUEUE(Insert(c, i), j)** ≻ **?rhs**$_2$

(3) **?rhs**$_2$ contains only the permitted operations of the implementing types, and the implementing functions.

The **Synthesis Theorem** in chapter 4 shows that, when a preliminary implementation is well-spanned, the preliminary implementation satisfies the homomorphism property if the synthesis equation corresponding to each of the rules in the preliminary implementation is a theorem of **PW**. Note that the second condition above ensures that the rewrite rules derived satisfy the uniform termination property. The third condition ensures the syntactic correctness of the preliminary implementation.

---

4. Note that since $\mathcal{H}$ is a function that behaves essentially like $\mathcal{A}$, the rewrite rules specifying it in **PW** are obtained by simply replacing $\mathcal{A}$ by $\mathcal{H}$ in the asociation specifcation.

2.2.1.3 Deriving the Synthesis Equations

Every synthesis equation of the preliminary implementation is derived with the help of two inference rules called the *synthesis rules.* The synthesis rules are designed for generating theorems of PW that have the same left hand sides, but different right hand sides. For deriving a synthesis equation, the synthesis rules are invoked repeatedly a finite number of times to generate a series of theorems until the desired equation is generated. For instance, the synthesis equation corresponding to the rule **ENQUEUE(Insert(c, i), j)** → $?\mathbf{rhs}_2$ is derived by generating a series of theorems that have **ℋ(ENQUEUE(Insert(c, i), j))** as their left hand side. The generation continues until a theorem whose right hand side qualifies the theorem to be a synthesis equation is encountered.

The idea used for generating an equation is to reverse the method of demonstrating that such an equation is a theorem of PW. The central notion used in the generation is a mechanism called *expansion.* Expansion[5] is the opposite of reduction. It is the act of applying a rewrite rule to an expression from right to left.

For example, consider the rule **ℋ(ENQUEUE(c, j))** → **Enqueue(ℋ(c), ℋ(j))**, and the expression **Add_at_head(Enqueue(ℋ(Create), ℋ(i)), k)**. The subexpression **Enqueue(ℋ(Create), ℋ(i))** is an instance of the right hand side of the rule for the substitution **{ c ↦ Create, j ↦ i }**. The corresponding instance of the left hand side is **ℋ(ENQUEUE(Create, i))**. Therefore, **Add_at_head(Enqueue(ℋ(Create), ℋ(i)), k)** expands to **Add_at_head(ℋ(ENQUEUE(Create, i)), k)** by the rule.

The first synthesis rule specifies a way of generating a theorem from an expression with that expression as the left hand side. In the following **e↓** denotes the normal form of **e** obtained using PW.[6] (The normal form of **e** is the result of reducing it using the rewrite rules of PW until it becomes irreducible.)

---

5. The definition of expansion will be revised later in chapter 4 to make it more general. According to the definition given here, expansion is identical to the transformation technique *folding* used by Darlington [7] for synthesis of recursive programs.

6. PW is a convergent system. Therefore, every expression is guaranteed to have a unique normal form.

Rule 1: $$\frac{\text{e is an expression}}{\mathbf{e \equiv e\downarrow}}$$

The second synthesis rule specifies how to generate a theorem from an existing one so that the new theorem has the same left hand side as the old one. In the following **expand**($\mathbf{e_2}$) denotes any expression that is an expansion of $\mathbf{e_2}$ using some rewrite rule of **PW**.

Rule 2: $$\frac{\mathbf{e_1 \equiv e_2}}{\mathbf{e_1 \equiv expand(e_2)}}$$

We investigate two methods in which the synthesis rules can be used for deriving a synthesis equation. The first method derives synthesis equations that are in the equational theory of **PW**. The second method derives equations that are in the inductive theory. The second method is more general than the first one. A system that implements the synthesis procedure would, therefore, use only the second method. We discuss them separately for pedagogic reasons.

2.2.1.3.1 Derivation in the Equational Theory

As an illustration, let us derive a synthesis equation that is of the form $\mathfrak{H}$(**ENQUEUE(Insert(c, i), j)**) $\equiv$ $\mathfrak{H}$(**?rhs**$_2$). The equation is derived by generating a series of theorems that have $\mathfrak{H}$(**ENQUEUE(Insert(c, i), j)**) as their left hand side. The generation is begun by invoking synthesis rule (1) on the left hand side expression. The rest of the theorems in the series are generated by invoking synthesis rule (2) using the rewrite rules of **PW** for expansion. The rewrite rules for expansion are chosen with the following ultimate goal: Obtain a right hand side that has the form $\mathfrak{H}$(**?rhs**$_2$) so that $\mathfrak{H}$(**ENQUEUE(Insert(c, i), j)**) $\succ$ $\mathfrak{H}$(**?rhs**$_2$), and **?rhs**$_2$ contains only the implementing functions and the permitted operations of the implementing types. In the illustration given below, the generation of every theorem in the series is considered as a step. At each step, the expression expanded, and the rewrite rule used for expansion are indicated. The relevant rewrite rules of **PW** that are going to be used for expansion are listed at the beginning. Rule (1) is the $\mathfrak{H}$-rule coresponding to **Enqueue**; rules (2) through (5) are obtained from the association specification.

Relevant Rewrite Rules of the Perturbed World

(1) $\mathcal{H}$(ENQUEUE(c, j)) → Enqueue($\mathcal{H}$(c), $\mathcal{H}$(j))

(2) $\mathcal{H}$(Create) → Nullq

(3) $\mathcal{H}$(Insert(c, i)) → Add_at_head($\mathcal{H}$(c), $\mathcal{H}$(i))

(4) Add_at_head(Nullq, i) → Enqueue(Nullq, i)

(5) Add_at_head(Enqueue(q, i), j) → Enqueue(Add_at_head(q, j), i)

Form of the theorem to be generated: $\mathcal{H}$(ENQUEUE(Insert(c, i), j)) ≡ $\mathcal{H}$($?rhs_2$)

Normal form of $\mathcal{H}$(ENQUEUE(Insert(c, i), j)): Enqueue(Add_at_head($\mathcal{H}$(c), $\mathcal{H}$(i)), $\mathcal{H}$(j))

Rules used for the normal form: (1), (3)

Step (1) Invoke Synthesis Rule (1) on $\mathcal{H}$(ENQUEUE(Insert(c, i), j))

$\mathcal{H}$(ENQUEUE(Insert(c, i), j)) ≡ Enqueue(Add_at_head($\mathcal{H}$(c), $\mathcal{H}$(i)), $\mathcal{H}$(j))

Step (2) Expand Expression: Enqueue(Add_at_head($\mathcal{H}$(c), $\mathcal{H}$(i)), $\mathcal{H}$(j))

Using Rule: (5)

$\mathcal{H}$(ENQUEUE(Insert(c, i), j)) ≡ Add_at_head(Enqueue($\mathcal{H}$(c), $\mathcal{H}$(j)), $\mathcal{H}$(i))

Step (3) Expand Expression: Enqueue($\mathcal{H}$(c), $\mathcal{H}$(j))

Using Rule: (1)

$\mathcal{H}$(ENQUEUE(Insert(c, i), j)) ≡ Add_at_head($\mathcal{H}$(ENQUEUE(c, j)), $\mathcal{H}$(i))

Step (4) Expand Expression: Add_at_head($\mathcal{H}$(ENQUEUE(c, j)), $\mathcal{H}$(i))

Using Rule: (3)

$\mathcal{H}$(ENQUEUE(Insert(c, i), j)) ≡ $\mathcal{H}$(Insert(ENQUEUE(c, j), i))

The theorem generated in step (4) qualifies to be a synthesis equation. Hence the desired rule of the preliminary implementation is:

ENQUEUE(Insert(c, i), j) → Insert(ENQUEUE(c, j), i)

One can similarly generate a theorem of the form $\mathcal{H}$(ENQUEUE(Create, j)) ≡ $\mathcal{H}$(Insert(Create, j)), which gives rise to the following rewrite rule to complete the preliminary implementation for

ENQUEUE:

ENQUEUE(Create, j) → Insert(Create, j)

2.2.1.3.2 Derivation in the Inductive Theory

The method used for deriving a synthesis equation in the inductive theory is based on the following property that every theorem of PW satisfies: If an equation is a theorem of PW, then every instance of it is in the equational theory of PW. An instance of an equation $e_1 \equiv e_2$ is an equation obtained by replacing every variable in e_1 and e_2 by generator constants.[7]

We, therefore, take the following approach for deriving an equation in the inductive theory. First derive an instance of the desired equation; the method of derivation described earlier can be used for this purpose. The instance of the equation derived should be such that a generalization of it has the form of the desired synthesis equation, and is a theorem of PW. A generalization of $e_1 \equiv e_2$ is an equation obtained by replacing assorted constants in e_1 and e_2 by suitable variables. To check if the generalization is a theorem of PW, we use an automatic procedure called **is-an-inductive-theorem-of**. The procedure is an extension of the method of using the Knuth-Bendix completion algorithm for proving inductive properties of convergent rewriting systems [28, 38, 22]. The procedure is described in chapter 4.

As an illustration let us derive a synthesis equation of the form $\mathcal{H}$(APPEND(c, Insert(d,i))) ≡ $\mathcal{H}$(?rhs$_2$) which gives rise to one of rules in the preliminary implementation of **Append**. We begin by deriving an instance determined by the replacement of the variable **d** by the constant **Create**, and then apply generalization.

Relevant Rewrite Rules of the Perturbed World

(10) **Append(q, Nullq) → q**
(14) **$\mathcal{H}$(Create) → Nullq**

---

7. A generator constant is an expression formed out of generators, and does not contain any variables.

(20) $\mathcal{H}$(ENQUEUE(c, i)) → Enqueue($\mathcal{H}$(c), $\mathcal{H}$(i))
(22) $\mathcal{H}$(APPEND(c, d)) → Append($\mathcal{H}$(c), $\mathcal{H}$(d))

Form of the theorem to be generated: $\mathcal{H}$(APPEND(c, Insert(Create, i))) ≡ $\mathcal{H}$(?e)
Normal form of $\mathcal{H}$(APPEND(c, Insert(Create, i))): Enqueue($\mathcal{H}$(c), $\mathcal{H}$(i))
Rules used for the normal form:

Step (1) Invoke Synthesis Rule (1) on $\mathcal{H}$(APPEND(c, Insert(Create, i)))
$\mathcal{H}$(APPEND(c, Insert(Create, i))) ≡ Enqueue($\mathcal{H}$(c), $\mathcal{H}$(i))

Step (2) Expand Expression: $\mathcal{H}$(APPEND(c, Insert(Create, i)))
Using Rule: (10)

$\mathcal{H}$(APPEND(c, Insert(Create, i))) ≡ Append(Enqueue($\mathcal{H}$(c), $\mathcal{H}$(i)), Nullq)

Step (3) Expand Expression: Nullq
Using Rule: (14)

$\mathcal{H}$(APPEND(c, Insert(Create, i))) ≡ Append(Enqueue($\mathcal{H}$(c), $\mathcal{H}$(i)), $\mathcal{H}$(Create))

Step (4) Expand Expression: Enqueue($\mathcal{H}$(*c*), $\mathcal{H}$(*i*))
Using Rule: (20)

$\mathcal{H}$(APPEND(c, Insert(Create, i))) ≡ Append($\mathcal{H}$(ENQUEUE(c, i)), $\mathcal{H}$(Create))

Step (5) Expand Expression: Append($\mathcal{H}$(ENQUEUE(c, i)), $\mathcal{H}$(Create))
Using Rule:

$\mathcal{H}$(APPEND(c, Insert(Create, i))) ≡ $\mathcal{H}$(APPEND(ENQUEUE(c, i), Create))

Step (6) Generalize the theorem in step (5) by replacing the constant
Create by the variable d to obtain the following equation:
$\mathcal{H}$(APPEND(c, Insert(d,i))) ≡ $\mathcal{H}$(APPEND(ENQUEUE(c, i), d))

Apply **is-an-inductive theorem-of** on the above equation.
This yields **True** *confirming that the equation is a theorem.*

Hence the desired rule (obtained by dropping $\mathcal{H}$ on both sides) is:

APPEND(c, Insert(d,i)) → APPEND(ENQUEUE(c, i), d)

One can similarly generate a theorem of the form $\mathcal{H}$(APPEND(Create, d)) ≡ $\mathcal{H}$(d) which gives rise to the following rewrite rule to complete the preliminary implementation of APPEND.

APPEND(Create, d) → d

2.2.2 Stage2: Derivation of the Target Implementation

In the second stage of the synthesis procedure, the preliminary implementation is transformed into a target implementation. It should be noted that the preliminary implementation is itself an executable implementation. It can be executed by an interpreter that is capable of simplifying algebraic expressions using the equations in the specifications of data types as rewrite rules. The data type verification system AFFIRM [39] provides such an interpreter. Given the specifications of all the implementing types, the interpreter can execute the preliminary implementation on any given input. Our goal is to derive the target implementation in a form that can be compiled by a compiler for an applicative language. There are two reasons why a target implementation is more efficient than a preliminary implementation. The first one arises because of the freedom to use nongenerators of the representation type in a target implementation. This makes it possible, in some instances, to eliminate recursion from a preliminary implementation of an operation, and to transform into one which is a composition of the operations of the implementing types. The second reason is that an implementation that can be compiled by means of a conventional compiler is in general more efficient than interpreting a set of rewrite rules. We investigate two methods of transforming a preliminary implementation into a target implementation. We describe each of them briefly below. The first method, although less efficient than the second, derives a larger set of implementations.

2.2.2.1 Recursion Eliminating Method

According to this method the problem of deriving a target implementation is viewed as finding a composition **f*** of the operations of the implementing types and the implementing functions (possibly including the **if_then_else** function) that has the same functional behavior as the implementing function F defined by the preliminary implementation. For example, the composition **Rotate(Insert(d, k))** has the same behavior as the function ENQUEUE defined by the rewrite rules of the following preliminary implementation:

ENQUEUE(Create, j) → Insert(Create, j)

ENQUEUE(Insert(c, i), j) → Insert(ENQUEUE(c, j), i)

So, the following can be a target implementation for it: **ENQUEUE(d, k) ::= Rotate(Insert(d, k))**. Note that the target implementation does not use recursion.

More formally, the problem can be stated as follows: Find a composition **f*** so that the equations obtained by substituting **f*** for ENQUEUE in the rewrite rules are theorems of the implementing types. The equations for ENQUEUE are given below. Note that, in obtaining the following equations, the two sides of the rewrite rules are interchanged after replacing ENQUEUE by **f***. The need for the interchange will be explained later.

(1) **Insert(Create, j) ≡ f*(Create, j)**

(2) **Insert(f*(c, j), i) ≡ f*(Insert(c, i), j)**

We use the following strategy to find a solution for **f***. We generate a theorem of the implementing types using one of the above equations as a template. For generating such a theorem we use the synthesis rules mentioned earlier. However this time, since we are interested in the theorems of the implementing types, the rewrite rules in the specification of the implementing types are used for expansion. The theorem generated determines a candidate for **f***. The goal is to generate a theorem so that the candidate for **f*** determined by the theorem also satisfies the other equation. For instance, the sequence of steps given below generates a theorem that has the form of equation (1).

Rewrite Rules of Circ_List

.....

(3) **Rotate(Create) → Create**

(4) **Rotate(Insert(Create, i)) → Insert(Create, i)**

(5) **Rotate(Insert(Insert(c, i1), i2)) → Insert(Rotate(Insert(c, i2)), i1)**

.....

Form of the theorem to be generated: **Insert(Create, j) ≡ f*(Create, j)**

Normal form of **Insert(Create, j)**: **Insert(Create, j)**

Rules used for the normal form: None

Step (1) Invoke Synthesis Rule (1) on **Insert(Create, j)**

Insert(Create, j) ≡ Insert(Create, j)

Step (2) Expand Expression: **Insert(Create, j)**

Using Rule: **(4)**

Insert(Create, j) ≡ Rotate(Insert(Create, j)

The last theorem generated in the above series suggests that **Rotate(Insert(d, k))** is a candidate for **f*(d, k)**. The candidate composition can be determined mechanically by comparing the theorem generated with the template equation. The candidate we currently have is such that the equation **Rotate(Insert(Insert(c, i), j)) ≡ Insert(Rotate(Insert(c, j)), i)**, which is obtained by replacing **f*** by **Rotate ∘ Insert** in equation (2), is a theorem of **Circ_List**. Had the candidate obtained in the last step not satisfied equation (2), the theorem generation would have continued further to generate another theorem that had the form of equation (1).

The reason that the first equation, rather than the second, was used as the template equation is the following. The synthesis rules are formulated so that the unknown expression in the equation to be searched for is on the right hand side. In equation (2) both sides are unknown since **f*** occurs on both the sides. That is not the case with equation (1). This was also the reason for interchanging the two sides of the rewrite rules while obtaining the template equations. In the example illustrated the theorem desired was in the equational theory. In general, we need to use the generalization technique described earlier since the

theorem may be in the inductive theory.

2.2.2.2 The Recursion Preserving Method

In this method the target implementation is derived with the help of a special set of functions, called the *inverting functions*,[8] on the representation type. To understand what inverting functions are, and why there are useful, let us consider an example. The preliminary implementation of **SIZE** consists of the following rules:

SIZE(Create) → 0

SIZE(Insert(c, i)) → SIZE(c) + 1

A target implementation for **SIZE** may take the following form:

SIZE(d) ::= if Empty(d) then 0
else SIZE(Remove(d)) + 1

Note that in the preliminary implementation the argument to **SIZE** on the left hand side of a rule is permitted to be a generator expression. The argument indicates the pattern or the structure of the expression that constructs the values for which the rewrite rule is applicable.[9] This freedom is used in a preliminary implementation to perform a case analysis based on the structure of the argument, and to decompose the argument.

In a target implementation the argument to **SIZE** on the left hand side of the definition must be a variable. This means that the expression on the right hand side of the definition must have explicit subexpressions for determining the structure of the argument, and to decompose the argument. Inverting functions of a data type can be used to build these subexpressions.

Informally speaking, the inverting functions of a data type are functions that can be

---

8. Inverting functions are closely related to *distinguished functions* of a data type defined in [24]. In [24] the distinguished functions are used to formalize the expressive power of a data type.
9. If we are interested in interpreting the preliminary implementation, it is, therefore, necessary for the interpreter to have pattern matching capability to invoke the appropriate rewrite rule while simplifying an expression.

used to algorithmically invert the process of constructing a value of the type from the generators of the type. In other words, by applying one or more of the inverting functions a finite number of times on a value one can determine a generator expression that constructs the value. For instance, for **Circ_List** the operations **Rotate**, **Value**, and **Empty** can serve as a set of inverting functions. The structure of any circular list value in terms of **Create** and **Insert** can be determined using these operations. For instance, if **v** is a variable denoting the value constructed by **Insert(c, j)**, then **Remove(v)** extracts the component **c**; **~Empty(v)** checks if **v** is constructed by an expression of the form **Insert(c, j)**. So, the rewrite rules can be merged into the following conditional expressions: **if Empty(d) then 0 else SIZE(Remove(d))+1.**

The target implementation is derived in two steps. The first step identifies a set of inverting functions for the representation type. In the second step the rewrite rules constituting the preliminary implementation of every operation are transformed into a target implementation in terms of the inverting functions. The method is described in detail in chapter 6.

2.2.3 Extending the Synthesis Procedure

Consider the association specification given in Fig. 6. It specifies a representation scheme for implementing **Queue_Int** as a triple **Array_Int X Integer X Integer**, which can informally be described as follows. (**Array_Int** is specified in the next chapter which also describes the association specification shown below in more detail.) **Nullq** can be represented

---

Fig. 6. Queue_Int in terms of Triple

$\mathcal{A}(\langle v, i, i\rangle) \equiv$ **Nullq**
$\mathcal{A}(\langle$**Assign(v, e, j)**$, i, j+1\rangle) \equiv$ **if** $i = j+1$ **then Nullq**
else Enqueue($\mathcal{A}(\langle v, i, j\rangle)$**, e)**

$\mathfrak{I}(\langle v, i, i\rangle) \equiv$ **True**
$\mathfrak{I}(\langle$**Assign(v, e, j)**$, i, j+1\rangle) \equiv$ **if** $i = j+1$ **then True**
else if $i \leq j+1$ **then** $\mathfrak{I}(\langle v, i, j\rangle)$
else False

by any triple in which the two integer components are equal. A nonempty queue can be represented by a triple **<v, i, j>**, where **v** is an array of arbitrary length containing the elements of the queue between the index values **i** and **j-1**, in order. In other words, **i** points to the front end of the queue, and **j** points to the next position available in the queue for adding an element. Note that in this example, unlike the last one, not every value of the representation type can legally represent a queue. A triple **<v, i, j>** is a legal representation value if only if $i \leq j$, and **v** is guaranteed to be defined on all index values between **i** and **j-1**. The invariant $\mathfrak{I}$ in Fig. 6 specifies this condition.

The synthesis the presence of a nontrivial invariant $\mathfrak{I}$ has to be performed differently because the implementation must be such that every implementing function **F** defined preserves $\mathfrak{I}$: That is, $(\forall v)[\mathfrak{I}(v) \Rightarrow \mathfrak{I}(F(v))]$.

The synthesis procedure for such a situation is similar to the one described earlier except for the method employed in determining the right hand sides of the rules of a preliminary implementation. The difference lies in the set of rewrite rules used for expansion while generating the theorems. Earlier, the rewrite rules of **PW** were used, but now it is necessary to use an additional set of rewrite rules. The additional rewrite rules describe information pertaining to the invariant $\mathfrak{I}$, and the assumption that the arguments to the implementing function satisfy the invariant. The information pertaining to $\mathfrak{I}$ is maintained as a separate entity called the *Temporary World*. Chapter 5 describes how the Temporary World is constructed, maintained, and used in the synthesis of an implemenation.

2.3 The Scope of the Synthesis Procedure

The scope of the synthesis procedure is limited because of two reasons. Firstly, the restrictions imposed on the input specifications limit the range of data type specifications that are acceptable as inputs to the procedure. Secondly, the synthesis procedure is capable of deriving only a class of implementations that satisfy certain properties. We describe the two forms of limitations below.

2.3.1 Restrictions on the Inputs

The input specifications must be such that the Initial World (IW), which is a combination of all the specifications, forms a rewriting system that

(1) has the uniform termination property,

(2) has the unique termination property, and

(3) is well-spanned.

The second and the third properties are not restrictive because they can be attained by adding certain additional rewrite rules to the system. There are automatic procedures [28, 38, 22] for determining the rules that need to be added, provided the system satisfies the uniform termination property.

The uniform termination property can be restrictive. It is, in general, not possible to express all the properties one wishes to specify in a manner that preserves the uniform termination property. For example, consider the data type **Set_of_Elements** that has an operation **Insert** to insert an element into a set. To express the property that the order of insertion of elements into a set is immaterial, it is necessary to have a rewrite rule of the form **Insert(Insert(s, i), j) → Insert(Insert(s, j), i)** as a part of IW. A system containing this kind of rule need not, in general, terminate because the rule does not strictly reduce an expression.

One way of getting around this problem is to exclude the concerned rule(s) from IW. However, there are two reasons why one may not want to do this. Firstly, the rule might be needed to attain the second and the third properties mentioned above. In such a situation excluding the rule(s) makes the input unacceptable. The second reason is that omitting the rule may leave the specification incomplete.[10] The method used by the synthesis procedure does not require the specifications to be complete, so the input (excluding the concerned rule) in this case is acceptable. But the procedure will not be able to derive an implementation that is dependent on the property expressed by the rule.

---

10. We use the following notion of completeness: A specification is complete if all the properties that are valid for the data type are provable from the specification.

2.3.2 The Class of Implementations Derived

There are three factors that are responsible for limiting the class of implementations derived by the procedure. The first is related to the subset of the proof theory of the input specifications in which the synthesis procedure operates. The procedure can only derive those implementations whose correctness proof is within the operational part of the theory. The operational part of the theory comprises the subset of the inductive theory that is decided by the Musser/Knuth-Bendix method [38] of proving inductive properties.

The second limiting factor is the termination ordering $\succ$. The synthesis procedure assumes that an effective ordering is implicitly available to be used in ensuring the termination of the implementation. So, the procedure can only derive those implementations whose termination can be proved using the ordering $\succ$. The more general[11] the ordering $\succ$, the larger is the class of implementations that can be derived.

The third reason is that the implementations derived may not involve arbitrary helping functions. The synthesis procedure is not capable of automatically discovering a helping function that might be necessary in an implementation. The user has to furnish a specification of the helping function as a part of the Initial World if he wishes an implementation in terms of the helping function.

2.3.3 Effects of Using the Procedure Outside its Scope

Using the procedure on a specification that does not satisfy the uniform termination property may result in infinite looping. This is because, under such a circumstance, there can be expressions for which a normal form does not exist. The effect of a violation of the unique termination property depends on how serious the violation is. If the violation implies that the system is inconsistent, then the procedure may derive an incorrect implementation. However, if the system is consistent despite the violation, the effect will only be a reduction in the class of implementations that the procedure can derive. It should be noted that all three of the

---

11. An ordering $\succ_1$ is considered to be more general [23] than $\succ_2$ if $\succ_1$ contains $\succ_2$. That is, $\succ_1$ relates a larger set of expressions than $\succ_2$.

properties required of the inputs can be checked automatically (assuming that a termination ordering $\succ$ is available).

3. Inputs to the Synthesis Procedure

This chapter has four sections. The first section defines data types and their specification. The second section describes the association specification. The third section characterizes the restrictions on the inputs. The last section describes proving properties of data types from the specifications.

3.1 Data Types and their Specification

3.1.1 Preliminary Concepts

A data type consists of a set (perhaps infinite) of values, called the *value set*, and a finite set of operations, called the *operation set*. The only way in which the values of a data type can be constructed, manipulated or observed is through the operations of the data type.

The behavior of a data type is usually dependent on several other data types. These data types appear as a part of the domain or as the range of the operations of the data type under consideration. We call these other data types the *defining types*; the data type under consideration is referred to as the *type of interest (TOI)*. If the TOI is the one that is being implemented, we refer to it as the *implemented type*. The type that is used to represent the implemented type is called the *representation type*. The defining types of the representation type are called the *ancillary types*. The union of the representation type and the ancillary types is called the set of *implementing types*. For example, the defining types of the data type **Queue_Int** specified in Fig. 7 are **Integer** and **Bool**.

A data type has two kinds of operations. A *constructor* is an operation that yields a value of the TOI, and an *observer* is an operation that yields a value of a defining type. For **Queue_Int**, the operations **Nullq**, **Enqueue**, **Dequeue**, and **Append** are all constructors; the rest of the operations are observers.

We treat the exceptional behavior of a data type in a simplified fashion. We assume that every data type has a unique exceptional value that is constructed by the operation **Error** belonging to the type. The value **Error( )** is treated like any other value of the type except that it has the following unique property. Every operation is assumed to be *strict* with respect

to **Error( )**: Every operation **f** is such that when applied to **Error( )** from any of its domain types it yields the exceptional value of the range type of **f**. We assume that every operation **f** is a total function: That is, **f** is defined on every element of its domain yielding either an exceptional value or a normal value from its range type.

The requirement on a data type that its values be manipulated only by its operations translates to requiring that its values be constructed only by its constructors, possibly using the values of its defining types. Furthermore, in a computer the values can be constructed only by a finite sequence of operations, so the value set of a data type is the smallest set closed under finitely many applications of its constructors. This property of a data type is called the *minimality property* [25].

A subset of constructors is said to be *complete* if every value of the TOI can be constructed by some composition of the constructors in the subset (possibly using values of the defining types). A *basis* for a data type is a complete set of constructors that is minimal, i.e., no subset of a basis is complete. A data type may have more than one basis. **{ Nullq, Enqueue }** is a basis for **Queue_Int** since all queues can be generated using **Nullq** and **Enqueue**, and no subset of it can do so.

An *expression* (or a *term*) is a sequence of operations and variables denoting an application of the operations to the variables. The *type of* an expression is the range type of the operation symbol that appears at the outermost level of the expression. A *constant* is an expression that does not contain any variables. For example, **Dequeue(Enqueue(q, e))** is an expression of type **Queue_Int**; it is not a constant since it contains variables. **Dequeue(Enqueue(Nullq, 0))** is a constant of type **Queue_Int**.

3.1.2 Definition of a Data Type

The only way in which the values of a data type can be manipulated is through the operations of the type. We define a data type so as to capture the behavior of the type as viewed through the operations of the type. This behavior is called the *observable behavior* of the data type. This method of definition was advocated by Guttag [16], and later developed by Kapur [25]. According to this view, the values of a data type are distinguishable only by

means of the operations of the type.

Heterogeneous algebras provide a natural means of modeling the behavior of a data type. A heterogeneous algebra that can be used to model a data type is defined recursively in terms of the algebra that is used to model each of its defining types. The basis of this recursion is the type **Bool** which does not have any defining types.

A heterogeneous algebra for a data type **D**, consists of (i) a domain corresponding to **D**, which is called the *principal domain*, (ii) a domain corresponding to every defining type of **D**, (iii) a function corresponding to every operation of **D**. The elements of the principal domain are used to denote the values of **D**. The minimality property of a data type requires that every element of the domains of the algebra be constructible by a finite number of applications of the constructors of the appropriate type. Any heterogeneous algebra that has the appropriate signature, and that exhibits the desired observable behavior can be used to model the data type. Hence, we define a data type as a set of heterogeneous algebras that exhibit the same observable behavior. Every algebra in the set is said to be a *model* of the data type. The elements of the principal domain are called the *values* (of **D**) in that model. Below we formally characterize the observable behavior of a heterogeneous algebra.

The observable behavior of a model is characterized in terms of the *distinguishability* relation on the values of the model. The distinguishability relation is defined inductively in terms of the distinguishability of the values of the defining types. That is, we assume that the distinguishability relation is already defined the domain corresponding to each of the defining types. (The basis of this induction is the data type **Bool** that does not have any defining types; the only two values, **True** and **False** of **Bool** are assumed to be distinguishable.) Two values of a model are distinguishable if and only if there is a sequence of operations of **D** with an observer as the outermost operation, that produces distinguishable results when applied separately on the values. If two values are not distinguishable, they are *observably equivalent*. For instance, the **Queue_Int** values constructed by **Enqueue(Nullq, 0)** and **Append(Nullq, Enqueue(Nullq, 0))** are observably equivalent; but the ones constructed by **Enqueue(Nullq, 0)** and **Dequeue(Enqueue(Nullq, 0))** are distinguishable. Observable equivalence is an equivalence relation.

Definition Two models are *behaviorally equivalent* if their quotient models induced by the observable equivalence relations are isomorphic to each other.

Definition A data type is a set of behaviorally equivalent heterogeneous algebras.

3.1.3 Specification of a Data Type

The specification of a data type is a piece of text in a formal language. It describes a set of properties concerning the operations of the data type. The aim of writing a specification is to characterize through the specification the observable equivalence relation that defines the data type.

It has been observed [17] that the construction of an algebraic specification for a data type is rendered easier and more reliable (in the sense that one has increased confidence in the consistency and completeness of the specification) by using a basis of the data type as a guide for constructing the specification. We assume that all our specifications are constructed in this fashion. The operations belonging to the basis of a specification are called the *generators* of the specification. An operation that is not in the basis is called a *non-generator.* Note that all generators are constructors; non-generators may be constructors or observers.

Throughout the development when we refer to the basis or the generators of a data type involved in the synthesis, we actually mean the basis or the generators associated with the specification of the data type being used as an input to the synthesis procedure. Definition of a couple of new terms pertaining to the generators are in order at this point. A *generator expression* (*generator constant*) of a data type is an expression (constant) that consists of only the generators of the type. Taking **Queue_Int** with the specification given in Fig. 7 as an example: **Enqueue(Nullq, 0)** is a generator constant whereas, **Dequeue(Enqueue(Nullq, 0))** is not a generator constant, because **Dequeue** is a non-generator.

3.1.3.1 The Specification Language

The specification language we use is a restricted version of an equational language that permits conditionals and auxiliary functions. The language is similar to the ones used in several other works on data type specification and verification such as [14, 18, 25]. A specification has two parts: the *Operations* part describes the functionality of every operation of the TOI; we assume that the *Operations* part identifies the basis used for the specification. The *Axioms* part consists of a set of axioms describing the properties of the operations. Every axiom has the form of an equation $\mathbf{e_1} \equiv \mathbf{e_2}$, where $\mathbf{e_1}$ and $\mathbf{e_2}$ are expressions of the same type. The expressions may involve any of the operations of the TOI and the defining types. The expressions may contain any of a finite number of auxiliary functions which are also specified as part of the specification. The equations may involve conditional expressions on their right hand side, i.e., $\mathbf{e_2}$ may contain the auxiliary function **if_then_else** which behaves like a conditional expression.[12] For the sake of clarity, we use the following more conventional syntax for an expression involving **if_then_else**. The expression **if_then_else(b,** $\mathbf{e_{21}}$, $\mathbf{e_{22}}$**)** is written as **if b then** $\mathbf{e_{21}}$ **else** $\mathbf{e_{22}}$.

We differ from the works cited above by assuming that every axiom in the specification satisfies the following syntactic constraints. The constraints are not restrictive, in the sense that they do not restrict the class of data types that can be specified. The first constraint enables us to automatically partition the axiom set into two disjoint sets: One that contains only the generator symbols; the other whose axioms may involve generators as well as nongenerators. The partitioning of the axiom set facilitates the synthesis process by reducing the inter-dependence of the synthesis of different operations. The second constraint permits the axioms to be treated as left to right rewrite rules (to be described later) without having to interchange the two sides of the axioms.

---

12. **if_then_else** can be specified by the following two equations.
if_then_else : Bool X T X T -> T
if_then_else(True, $\mathbf{e_1}$, $\mathbf{e_2}$**)** $\equiv \mathbf{e_1}$
if_then_else(False, $\mathbf{e_1}$, $\mathbf{e_2}$**)** $\equiv \mathbf{e_2}$

Every axiom $e_1 \equiv e_2$ of a specification satisfies the following conditions:

(1) Every data type specification explicitly identifies a basis, i.e., a set of generators.

(2) The set of variables in e_2 is a subset of the set of variables in e_1.

Figures 7 and 8 show specifications of a (FIFO) queue of integers (**Queue_Int**) and a circular list of integers (**Circ_List**). The specifications meet the constraints specified above.

3.1.3.2 Semantics of a Specification

The specification of a data type characterizes the observable equivalence relation that defines the data type. The semantics of a specification is a set of heterogeneous algebras that are behaviorally equivalent based on the observable equivalence relation characterized by the specification.

To determine the observable equivalence relation characterized by a specification, the symbol '≡' in the axioms of the specification should be read as 'observably equivalent'. For instance, the equation **Size(Enqueue(q, e)) ≡ Size(q) + 1** in the specification of **Queue_Int** asserts that the two expressions yield observably equivalent values for all instantiations of the variables in them. The observable equivalence relation characterized by the specification is the reflexive, symmetric, transitive closure of ≡. Every algebra that satisfies all the axioms in the specification is a model of the type being specified by specification.

3.2 Association Specification

In addition to the specifications of the types involved in the synthesis, the synthesis procedure expects the user to provide information about the representation scheme to be used by the implementation that is to be derived. This section explains what exactly that information is, and how it can be specified. We call the formal description of the information the *association specification* of an implementation.

Fig. 7. Specification of Queue_Int

Queue_Int is Nullq, Enqueue, Front, Dequeue, Append, Size

Defining Types

Bool, Int

Operations

Nullq : -> Queue_Int
Enqueue : Queue_Int X Int -> Queue_Int
Front : Queue_Int -> Int ∪ { ERROR }
Dequeue : Queue_Int -> Queue_Int ∪ { ERROR }
Append : Queue_Int X Queue_Int -> Queue_Int
Size : Queue_Int -> Int

Basis

{ Nullq, Enqueue }

Axioms

(1) Front(Nullq) ≡ ERROR
(2) Front(Enqueue(Nullq, e)) ≡ e
(3) Front(Enqueue(Enqueue(q, e1), e2)) ≡ Front(Enqueue(q, e1))

(4) Dequeue(Nullq) ≡ ERROR
(5) Dequeue(Enqueue(Nullq, e)) ≡ Nullq
(6) Dequeue(Enqueue(Enqueue(q, e1), e2)) ≡ Enqueue(Dequeue(Enqueue(q, e1)), e2)

(10) Append(q, Nullq) ≡ q
(11) Append(q1, Enqueue(q2, e2)) ≡ Enqueue(Append(q1, q2), e2)

(12) Size(Nullq) ≡ 0
(13) Size(Enqueue(q, e)) ≡ Size(q) + 1

Fig. 8. Specification of Circ_List

Circ_List is Create, Insert, Value, Remove, Rotate, Empty, Join

Defining Types

Integer, Boolean

Operations

Create : -> Circ_List
Insert : Circ_List X Integer -> Circ_List
Value : Circ_List -> Integer ∪ { ERROR }
Remove : Circ_List -> Circ_List ∪ { ERROR }
Rotate : Circ_List -> Circ_List
Empty : Circ_List -> Boolean
Join : Circ_list X Circ_list -> Circ_list

Comment

Circ_List is a list of integers with a front end and a rear end. **Create** constructs an empty list; the front and the rear ends of an empty list are the same. **Insert** inserts an element into a list at the rear end. **Value** returns the element at the rear end of a list. **Remove** removes the element at the rear end from a list. **Rotate** moves every element in a list by one position towards the rear end in a cyclic fashion, i.e., the element at the rear is moved to the front. **Empty** checks if a list is empty. **Join** joins two lists by positioning the first argument in front of the second.

Basis

{Create, Insert}

Axioms

(1) Value(Create) ≡ ERROR
(2) Value(Insert(c, i)) ≡ i

(3) Remove(Create) ≡ ERROR
(4) Remove(Insert(c, i)) ≡ c

(5) Rotate(Create) ≡ Create
(6) Rotate(Insert(Create, i)) ≡ Insert(Create, i)
(7) Rotate(Insert(Insert(c, i1), i2))) ≡ Insert(Rotate(Insert(c, i2)), i1)

(8) Empty(Create) ≡ true
(9) Empty(Insert(c, i)) ≡ false

(10) Join(c, Create) ≡ c
(11) Join(c, Insert(d, i)) ≡ Insert(Join(c, d), i)

3.2.1 What is an Association Specification ?

An association specification characterizes two pieces of information about a representation scheme:

(1) The set of values of the representation type that an implementation may use in representing the values of the implemented type. We call this set the *representing domain* ($\mathfrak{R}$). $\mathfrak{R}$ is characterized by means of a predicate on the representation type called the *invariant* ($\mathfrak{I}$): $\mathfrak{R}$ is the set of values of the representation type for which $\mathfrak{I}$ is **True**.

(2) A function, called the *abstraction function*, from the values of the representation type to the values of the implemented type. The function corresponds to the representation function of a data type introduced by [21]. The abstraction function maps a representation value to an abstract value that the former may represent in an implementation. An abstraction function may be a many-to-one function. An abstraction does not have to be defined on every value of the representation type. However, it has to be defined on every value in the representing domain.

The information characterized by the association specification is often the most creative part of an implementation. The proof of correctness of an implementation also, in general, needs to use information such as this. If the invariant part of an association specification is vacuous, then we assume that the invariant is true on all values of the representation type. In such a case the representing domain includes all the values of the representation type.

3.2.2 How Is It Expressed ?

We specify $\mathfrak{I}$ and $\mathcal{A}$ using the same language that is used to specify the data types involved. $\mathfrak{I}$ is specified as a set of equations, like any other predicate on the value set of the representation type. $\mathcal{A}$ is specified as a set of equations relating expressions of the representation type to expressions of the implemented type. We require that $\mathcal{A}$ be specified as a well-defined function with a nonempty domain.

Fig. 9. Two Association Specifications for Queue_Int

9(a) Queue_Int *in terms of* Circ_List

$\mathcal{A}$(Create) ≡ Nullq
$\mathcal{A}$(Insert(c, i)) ≡ add_at_head($\mathcal{A}$(c), i)

add_at_head(Nullq) ≡ Enqueue(Nullq, i)
add_at_head(Enqueue(q, i), i1) ≡ Enqueue(add_at_head(q, i1), i)

9(b) Queue_Int *in terms of* Array_Int X Int X Int

$\mathcal{A}$(<v, i, i>) ≡ Nullq
$\mathcal{A}$(<Assign(v, e, j), i, j+1>) ≡ if i = j+1 then Nullq
else Enqueue($\mathcal{A}$(<v, i, j>), e)

$\mathcal{I}$(<v, i, i>) ≡ True
$\mathcal{I}$(<Assign(v, e, j), i, j+1>) ≡ if i = j+1 then True
else if j+1 < i then False
else $\mathcal{I}$(<v, i, j>)

Fig. 9 gives a couple of example of an association specification. 9(a) specifies an implementation of **Queue_Int** in terms of **Circ_List**. The empty queue is represented by the empty list; a nonempty queue is represented by a list whose elements are identical to the ones in the queue, but are arranged in the reverse order. The motivation for this representation scheme is that reading and deletion of elements from a queue can be performed efficiently.

Consider the association specification given in Fig. 6. It specifies a representation scheme for implementing **Queue_Int** as a triple , which can informally be described as follows. (**Array_Int** is specified in the next chapter which also describes the association specification shown below in more detail.)

Fig. 9(b) specifies an implementation in which a queue is implemented as a triple **Array_Int X Integer X Integer**. (**Array_Int** is specified in Fig. 10.) The representation scheme can be informally described as follows. **Nullq** can be represented by any triple in which the two integer components are equal. A nonempty queue can be represented by a triple **<v, i, j>**,

Fig. 10. Specification of Array_Int

Array_Int is Nullarr, Assign, Read, Size, Empty

Defining Types

Integer, Boolean

Operations

Nullarr : -> Array_Int
Assign : Array_Int X Integer X Integer -> Array_Int
Read : Array_Int X Integer -> Integer ∪ { ERROR }
Size : Array_Int -> Integer
Empty : Array_Int -> Boolean

Comment

Array_Int is an array of integers. Every element in the array is indexed by an integer; the indices are not necessarily contiguous. **Nullarr** creates an empty array. **Assign** assigns a given value (the second argument) to the element at a given index (the third argument); if the array does not have an element with the given index, then the value is added to the array. **Read** reads the element at the given index. **Empty** checks if an array is empty.

Basis

{Nullarr, Assign}

Axioms

(1) Assign(Assign(v, e1, i1), e2, i2) ≡ if i1 = i2 then Assign(v, e2, i2)
else Assign(Assign(v, e2, i2), e1, i1)

(2) Read(Nullarr, i) ≡ ERROR
(3) Read(Assign(v, e, j), i) ≡ if i = j then e
else Read(v, i)

(4) Empty(Nullarr) ≡ true
(5) Empty(Assign(v, e, i)) ≡ false

where **v** is an array of arbitrary length containing the elements of the queue between the index values **i** and **j-1**, in order. In other words, **i** points to the front end of the queue, and **j** points to the next position available in the queue for adding an element.

Note that in this example, unlike the last one, not every value of the representation type can legally represent a queue. A triple **<v, i, j>** is a legal representation value if only if $\mathbf{i} \leq \mathbf{j}$, and **v** is guaranteed to be defined on all index values between **i** and **j-1**. The invariant $\mathcal{I}$ in specifies this condition.

The abstraction function $\mathcal{A}$ is specified so that it is defined on all values for which $\mathcal{I}$ is **True**. The specification uses an auxiliary function **Add_at_head**. **Add_at_head** is a function on **Queue_Int** that adds a given element at the front of a queue. A specification of **Add_at_head** is given as a part of the association specification.

3.2.3 Further Discussion on Association Specification

It is important to note that every association specification need not have an implementation corresponding to it. To understand this more clearly, let us look at the relationship between an association specification and an implementation that uses a representation scheme consistent with the one characterized by the association specification.

An implementation of a data type consists of

(i) a representation type being used as the representation for the implementation.

(ii) a program, i.e., a segment of code, for every operation of the type in a language; this program is called the implementation of the corresponding operation.

Note that both a preliminary implementation and a target implementation (as introduced in the previous chapter) of a data type are implementations of the data type. A preliminary implementation uses one language to express the program, while the target implementation uses another.

Formally, an implementation of a data type can be considered to be denoting a heterogeneous algebra, called an *implementation algebra*, with

(i) a principal domain that is a subset of the value set of the representation type,

(ii) a domain corresponding to every defining type of the implemented type - this domain is identical to the value set of the corresponding defining type,

(iii) a function corresponding to the implementation of every operation of the implemented type so that the function mimics the behavior of the implementing program.

An implementation of a type is correct if there exists a homomorphism, from the implementation algebra to to the implemented type. The association specification should be such that there exists an implementation algebra with computable functions that corresponds to the representation scheme characterized by the association specification. More specifically, the implementation algebra should satisfy the following conditions:

(i) The principal domain of the algebra is the representing domain characterized by the association specification.

(ii) There is a computable function in the algebra with the appropriate functionality corresponding to every operation of the implemented type.

(iii) The implemented data type is a homomorphic image of the implementation algebra with respect to the abstraction function.

We do not intend to formally characterize the properties that the association specification ought to satisfy so that it meets the above requirement. Rather, we trust the intuition of the user, and assume that there exists an implementation that is consistent with the association specification furnished by him. If the association specification provided as an input to the synthesis procedure is such that there is no implementation corresponding to it, then the synthesis procedure will, in general, never terminate. The synthesis method, however, does not produce an incorrect implementation in such a case.

3.3 Restrictions on the Inputs

The method used by the synthesis procedure to derive an implementation is based on treating every equation in the specifications as a rewrite rule. The procedure combines all the input specifications, and treats the union as a set of rewrite rules called the *Initial World.* The restrictions imposed on the inputs are intended to ensure that the Initial World satisfies a useful property called the *principle of definition.*

The first subsection informally introduces the basic concepts about rewrite rules. (See Appendix I for formal definitions.) The second subsection defines principle of definition, and develops a sufficient set of conditions for principle of definition (SCPD). The input is expected to satisfy SCPD. The third subsection describes how to prove properties from a specification that satisfies SCPD.

3.3.1 Rewrite Rules and Rewriting Systems

A rewrite rule is an ordered pair **(left, right)**, written **left** $\rightarrow$ **right**, where **left** and **right** are expressions containing variables so that the variables in **right** are among the variables in **left**. A rule is used to *reduce* an expression by replacing any subexpression that is matched by **left** with a corresponding version of **right**, i.e., with the same substitutions for variables that were made in matching **left**. (More precise definitions are given in Appendix I.)

For example, consider the rule $\mathbf{Append(q_1, Enqueue(q_2, i_2)) \rightarrow Enqueue(Append(q_1, q_2), i_2)}$, and the expression $\alpha = \mathbf{Dequeue(Append(q_3, Enqueue(Nullq, 0)))}$. α is *reducible* using the rule because it has a subexpression $\alpha' = \mathbf{Append(q_3, Enqueue(Nullq, 0))}$ that *has the form of* the left hand side of the rule: That is, $\mathbf{Append(q_1, Enqueue(q_2, i_2))}$ becomes identical to $\mathbf{Append(q_3, Enqueue(Nullq, 0))}$ when the variables in the former are substituted according to the *substitution* $\sigma = [\mathbf{q_1 \mapsto q_3, q_2 \mapsto Nullq, i_2 \mapsto 0}]$. The corresponding instance of the right hand side of the rule (obtained by substituting the variables in $\mathbf{Enqueue(Append(q_1, q_2), i_2)}$ using the substitution σ) is $\beta' = \mathbf{Enqueue(Append(q_3, Nullq), 0)}$. $\beta = \mathbf{Dequeue(Enqueue(Append(q_3, Nullq), 0))}$ is the expression obtained by replacing α' by β' in α. Then, we say that α *reduces to* β, written $\alpha \rightarrow \beta$.

A *rewriting system* is a set of rewrite rules. Let **R** be a rewriting system. An expression α is *reducible* by **R** if it is reducible by some rule in **R**. If α is not reducible by any rule in **R**, then α is *irreducible* by **R**.

If $\alpha \rightarrow \beta$ by a rule in **R**, then we say that α *directly reduces to* β using **R**, and once again write it as $\alpha \rightarrow \beta$ (using **R**). Let $\rightarrow^*$ be the smallest relation on pairs of expressions which is the reflexive, transitive closure of $\rightarrow$. Thus, $\alpha \rightarrow^* \beta$ if and only if there exist expressions $\alpha_0, \alpha_1, \ldots, \alpha_n$, where $n \geq 0$, such that $\alpha = \alpha_0$, $\alpha_i \rightarrow \alpha_{i+1}$ for $i = 0, \ldots, n\text{-}1$ and $\alpha_n = \beta$. We read $\alpha \rightarrow^* \beta$ as 'α *reduces to* β.

Suppose $\alpha \rightarrow^* \beta$, and β is irreducible. Then we say that α *simplifies* to β; β is called a *normal form* of α (in **R**).

Rewriting systems are used to simplify expressions into their normal forms. Thus, a useful property of a system is *uniform termination*: **R** has the uniform termination property if no infinite sequence of reductions, $\alpha_0 \rightarrow \alpha_1 \rightarrow \ldots$, is possible in **R**. When **R** has the uniform termination property every expression is guaranteed to have a normal form. Another useful property of a rewriting system is *unique termination*: **R** has the unique termination property if any two terminating sequences of reductions starting from the same expression have identical final expressions. When **R** has the unique termination property the normal form (if it exists) of every expression is unique. A rewriting system that has both the uniform termination property and the unique termination property is said to be *convergent*. When **R** is convergent every expression α has exactly one normal form; we denote the unique normal form of α in a convergent system by $\alpha\downarrow$.

The rewriting systems corresponding to our input specifications are obtained by simply replacing the symbol '$\equiv$' by the symbol '$\rightarrow$' in each of the equations in the specifications. For example, Fig. 11 gives the rewriting system corresponding to the specification of **Queue_Int** in Fig. 7. Henceforth, we treat the input specifications as rewriting systems obtained as explained above. When we refer to a specification, we actually mean the rewriting system obtained from the specification.

Fig. 11. The Queue_Int Rewriting System

(1) Front(Nullq) → ERROR
(2) Front(Enqueue(Nullq, e)) → e
(3) Front(Enqueue(Enqueue(q, e1), e2)) → Front(Enqueue(q, e1))

(4) Dequeue(Nullq) → ERROR
(5) Dequeue(Enqueue(Nullq, e)) → Nullq
(6) Dequeue(Enqueue(Enqueue(q, e1), e2)) → Enqueue(Dequeue(Enqueue(q, e1)), e2)

(10) Append(q, Nullq) → q
(11) Append(q1, Enqueue(q2, e2)) → Enqueue(Append(q1, q2), e2)

(12) Size(Nullq) → 0
(13) Size(Enqueue(q, e)) → Size(q) + 1

3.3.2 The Principle of Definition

The *principle of definition* is a property of a specification (or a group of specifications). The property ensures the consistency of a specification. The property reinforces the two-tier characteristic inherent in our specifications: It ensures that the generators are specified among themselves, and the nongenerators are specified as total functions in terms of the generators. Finally, the property is useful in mechanically proving properties of data types from their specifications. The property is similar to a property with the same name defined in [22]. Our definition is more general than the one in [22].

Definition *The Principle of Definition*

A specification (or a group of specifications) S has the *principle of definition* property if every constant t has exactly one normal form (in S), and the normal form is a generator constant of the appropriate type.

There will be situations in our development when it is necessary to use a restricted version of the principle of definition. The notion is restricted in the sense that the principle of definition need hold good only for a subset of terms. The restricted property is useful in stating that every nongenerator defined by a system be defined as a total function on a subset

of the value set of a type. We give a definition the property below.

Definition *Principle of Definition With Respect* T

Let T be a set of generator constants not necessarily including all possible constants. A system S satisfies the *principle of definition with respect to* T if the following condition holds: Every constant of the form $F(g_1, \ldots, g_n)$, where F is a nongenerator function symbol and $g_1, \ldots, g_n$ are generator constants in T, has a unique normal form (in S) that is a generator constant in T.

The principle of definition has two parts to it: It requires every constant to have a unique normal form in S, and the normal form to be a generator constant. SCPD has to be formulated so as to ensure the two parts. The first part can be ensured by requiring S to be convergent (i.e., to satisfy the uniform termination property and the unique termination property). The second part is ensured by requiring S to be *well-spanned.* We define what it means for S to be well-spanned below, and then show how the two properties ensure the principle of definition of S.

Consider the rewriting system shown in Fig. 11. The system has three rules (1, 2, and 3) in which the expression on the left hand side has **Front** as its outermost symbol. The set, **{Nullq, Enqueue(Nullq, e), Enqueue(Enqueue(q, e1), e2)}**, of generator expressions that appear as arguments to **Front** on the left hand side in the rules spans the entire set of generator constants of **Queue_Int**; in other words, every generator constant of type **Queue_Int** is an instance of one of the expressions in the above set. When a rewriting system has enough rules corresponding to a nongenerator function **f** so that the set of generator expressions appearing as arguments to **f** spans the set of all generator constants, we say that **f** is *well-spanned* by the rewriting system. We say that a rewriting system is *well-spanned* if every nongenerator function symbol of the system is well-spanned. We formalize this notion below.

In general, since **f** can be multi-ary, the arguments to **f** are k-tuples of expressions of appropriate types, where k is the arity of **f**. In the following formalization, we first define the notion of a set of k-tuple of generator expressions being *well-spanned*; informally, a set of

k-tuples of generator expressions is well-spanned if it spans the set of all k-tuples of generator constants of appropriate types. The property of a function being well-spanned is defined in terms of the notion of a well-spanned set of k-tuple of generator expressions. In the following, we assume that the k-tuples are homogeneous with regard to the types of their components. The extension to the heterogeneous case is simple.

Definition A set $A = \{A_1, \ldots, A_p\}$ of k-tuples of generator expressions $A_i = \langle e_{i1}, \ldots, e_{ik} \rangle$ is *well-spanned* if the following condition holds: For every k-tuple, $\langle t_1, \ldots, t_k \rangle$, of generator constants there exist n, $1 \leq n \leq p$, and a substitution σ, such that for every j, $1 \leq j \leq k$, we have $t_j = \sigma(e_{nj})$. .

Definition A nongenerator function **f** is *well-spanned* by a rewriting system **R** if there is in **R** a set of rewrite rules whose left hand sides are of the form $f(e_{i1}, \ldots, e_{ik})$, $1 \leq i \leq p$, and the set $\{ \langle e_{i1}, \ldots, e_{ik} \rangle \mid 1 \leq i \leq p \}$ is complete.

Definition A rewriting system **R** is *well-spanned* if every nongenerator function symbol in **R** is well-spanned.

Definition A specification **S** satisfies the *sufficient condition for the principle of definition (SCPD)* if **S** satisfies the following conditions:

(i) S is convergent

(ii) S is well-spanned.

Lemma If S satisfies SCPD then S satisfies the principle of definition.

Proof Condition (i) guarantees that every constant has exactly one normal form. Condition (ii) implies that every constant of the form $f(g_1, \ldots, g_n)$, where **f** is a nongenerator and $g_1, \ldots, g_n$ are generator constants is reducible. Since **S** satisfies uniform termination property, this means that no constant with a nongenerator can be a normal form. Hence the normal form of every constant is a generator constant.

Q.E.D

3.3.3 Checking the Principle of Definition

The main reason for formulating SCPD is so that we might be able to develop effective methods of checking if a specification satisfies the principle of definition. This section sheds some light on this topic.

To check if a specification is well-spanned, we have to check if the set of expressions (or k-tuples of expressions) that appear as arguments to each of the implementing functions is complete. Huet in [22] has demonstrated that it is possible to come up with an effective set of conditions that is sufficient to check if a set of expressions is complete.

Checking the convergence of a set of rules, which forms the remaining condition of SCPD, has been investigated in [28, 22]. The result in the cited works, which is due to Knuth and Bendix, provides an algorithm (henceforth referred to as the KB-algorithm) to check the convergence of a finite set of rewrite rules that satisfies the uniform termination property. Thus, if we can independently ensure the uniform termination property of a specification, then we can use the KB-algorithm to show the unique termination property of the specification.

3.3.3.1 Checking Unique Termination

Let **R** be a finite set of rewrite rules that has the uniform termination property. The following theorem is the basis for the KB-algorithm for checking the unique termination property. The theorem depends upon the concept of *unification* of expressions. We will first define this concept.

Two expressions α and β with disjoint variable sets are said to be *unifiable* if there exists a substitution θ such that $\theta(\alpha) = \theta(\beta)$.[13] The *most general unifier* of two unifiable expressions α and β is the unifier θ, such that for any unifier σ of α and β there exists a substitution ρ such that σ is the composition of ρ and θ. The *unification algorithm* of Robinson [44] can be used to determine a most general unifier of two given expressions or

---

13. The symbol $=$ stands for two expressions being identically equal.

decide that they are not unifiable. In the discussion to follow we assume that the candidates for unification have variables renamed if necessary to obtain disjoint variable sets.

Let $\gamma_1 \rightarrow \delta_1$ and $\gamma_2 \rightarrow \delta_2$ be two rules of **R** so that γ_1 is unifiable with a nonvariable subexpression of γ_2. More precisely, there exists an occurrence u in γ_2 such that $\alpha = \gamma_2/u$ is not a variable, and α is unifiable with γ_1. Let θ be the most general unifier of α and γ_1. Then, we say that $\theta(\gamma_2)$ is a *superposition* of γ_1 on γ_2. (If β is either a superposition of γ_1 on γ_2 or a superposition of γ_2 on γ_1, then we say that β is a superposition between γ_1 and γ_2.) To each superposition there corresponds a *critical pair* $\langle \alpha_1, \alpha_2 \rangle$ of expressions defined as follows. α_1 and α_2 are the expressions obtained by applying to $\theta(\gamma_2)$ the above two rules, respectively. More precisely,

$$\alpha_1 = \theta(\gamma_2)[u \leftarrow \theta(\delta_1)]$$

$$\alpha_2 = \theta(\delta_2)$$

For example, consider the following rules

Append(q1, Enqueue(q2, i2)) → Enqueue(Append(q1, q2), i2)

Append(Append(q3, q4), q5)) → Append(q3, Append(q4, q5))

γ_1 is unifiable with the entire expression γ_2 by the most general unifier θ = [Append(q3, q4) for q1, Enqueue(q2, i2) for q5], yielding the superposition α and the critical pair $\langle \alpha_1, \alpha_2 \rangle$ shown below:

α = **Append(Append(q3, q4), Enqueue(q2, i2))**

α_1 = **Enqueue(Append(Append(q3, q4), q2), i2)**

α_2 = **Append(q3, Append(q4, Enqueue(q2, i2)))**

Theorem 1 The KB-Theorem

If **R** has the finite termination property, then it has the unique termination property if and only if every critical pair $\langle \alpha_1, \alpha_2 \rangle$ of **R** has the property that α_1 and α_2 have identical normal form.

Proof For a proof see [28, 22].

If a finite rewriting system has no superpositions, and therefore, no critical pairs, it is said to

be *superposition-free.* Thus, we trivially have:

Corollary If a finite rewriting system has the uniform termination property, and is superposition-free, then it has the unique termination property.

For example, the rewriting system in Fig. 11 corresponding to **Queue_Int** is superposition-free. In the next subsection we show that it satisfies the uniform termination property. So the rewriting system is convergent.

3.3.3.2 Checking Finite Termination

A general technique for checking termination of a rewriting system **R** is to demonstrate that it is possible to define a well-founded partial ordering $\succ$ on the set of all constants (that can be constructed using the function symbols in **R**) so that $t_1 \rightarrow t_2$ implies $t_1 \succ t_2$. A partial ordering is well-founded if there are no infinite descending sequences such as $t_1 \succ t_2 \succ ...$ for any constants. Hence, there cannot be any infinite sequence of rewrites using **R** also. Appendix II goes into this topic in greater detail. It describes a theorem that provides a useful guideline to define a suitable partial ordering to check the uniform termination property of a rewriting system.

We assume that a well-founded partial ordering $\succ$ on expressions is available as an input to the synthesis procedure. The ordering $\succ$ is used by the synthesis procedure not only to ensure the uniform termination property of inputs, but also to ensure that the output synthesized terminates. The orderings used in our examples belong to a class of orderings, called the *lexicographic recursive path* ordering [26, 10]. A formal definition of the ordering is given in Appendix II.

3.4 Proving Properties of a Data Type

The properties of a data type we are interested in are always expressed as equations of the form $e_1 \equiv e_2$, where e_1 and e_2 are expressions, and $\equiv$ denotes the observable equivalence relation (see sec. 3.1.2). For instance, the property **Append(Append(q_1, q_2), q_3)** $\equiv$ **Append(q_1, Append(q_2, q_3))** asserts that for every instantiation of

the variables by values the expressions on the two sides of the equation yield observably equivalent values. Our objective is to prove a property as a *theorem* from a specification of the type. This is crucial to our work because synthesis of implementations involves searching for appropriate theorems of the input specifications. In the following, we describe how to mechanically prove theorems from a specification that satisfies the principle of definition.

Definition A Theorem of a Specification

Let S be a specification (or a group of specifications). Let σ be a substitution that maps variables to generator constants. An equation $\mathbf{e}_1 \equiv \mathbf{e}_2$ is a theorem of S if for every σ the constants $\sigma(\mathbf{e}_1)$ and $\sigma(\mathbf{e}_2)$ have identical normal forms.

Note that the above definition of a theorem gurantees that if $\mathbf{e}_1 \equiv \mathbf{e}_2$ is a theorem of S then $\mathbf{e}_1$ and $\mathbf{e}_2$ always yield observably equivalent values. This is because the principle of definition ensures that for every instantiation of the variables (in $\mathbf{e}_1$ and $\mathbf{e}_2$) by generator constants the two expressions simplify to the same generator constant. This provides a basis for developing a method for mechanically proving properties of data types from specifications.

Note that the reverse of the above implication is not true. This is because we require that the input specifications be only consistent (via the principle of definition), but not *complete* [25]. A specification S of a data type **D** is complete if every equation $\mathbf{e}_1 \equiv \mathbf{e}_2$ such that $\mathbf{e}_1$ and $\mathbf{e}_2$ are observably equivalent for **D** is a theorem of S. The synthesis procedure would be more productive if the input specifications are complete. This is because it is possible to prove more properties from a complete specification, and hence the synthesis procedure might be able to derive a larger class of implementations.

There are several ways in which the above result can be used to deduce that an equation is a theorem of a specification. The methods differ in the reasoning or logic used for the deduction. In our development we deal with two kinds of logic: the *equational logic*, and the *inductive logic*.

Equational Logic

In the equational logic $\mathbf{e}_1 \equiv \mathbf{e}_2$ is deduced to be a theorem of S by checking if $\mathbf{e}_1$ and

e_2 have the same normal form in S. Note that if $e_1\downarrow = e_2\downarrow$, then it is obvious that e_1 and e_2 have identical normal forms for every substitution of the variables by generator constants. ($e\downarrow$ denotes the normal form of e.) An equation deduced to be a theorem of S in this fashion is said to be a theorem in the equational theory of S. When S satisfies the principle of definition every expression is guaranteed to have a unique normal form. Therefore, it is possible to develop a general procedure to decide the entire equational theory of S. As an illustration, we give a proof of **Append(Append(q_1, q_2), Nullq) ≡ Append(q_1, Append(q_2, Nullq))** using the specification of **Queue_Int** shown in Fig. 11.

Equation to be proved: **Append(Append(q_1, q_2), Nullq) ≡ Append(q_1, Append(q_2, Nullq))**

Normal form of left hand side:

Append(Append(q_1, q_2), Nullq)

Rule (10) ↓

Append(q_1, q_2)

Normal form of right hand side:

Append(q_1, Append(q_2, Nullq))

Rule(10) ↓

Append(q_1, q_2)

Inductive Logic

A property Φ is deduced to be a theorem in the inductive logic by using, besides the reduces relation $\rightarrow^*$, some form of mathematical induction. A property that is deduced using the inductive logic is called a *theorem in the inductive logic.* The set of all properties that can be deduced from a specification using the inductive logic is called the *inductive theory* of the specification.

The induction used is carried over the set of all generator constants using one or more of the variables in Φ as parameters for the induction. The induction is based on any well-founded partial ordering on the set of generator constants. Suppose G is the set of all generator constants, and $\succ$ is a well-founded partial ordering on G. Suppose we are using the variable v of Φ(v) as the parameter of induction. Then the induction rule may be stated as follows:

Induction rule

If for every $t \in G$ we can show that, for every $t' \in G$ such that $t \succ t'$, $\Phi[v/t'] \Rightarrow \Phi[v/t]$, then $\Phi(v)$ is theorem.

To apply the induction rule, we have to define a partial ordering $\succ$ on G. Since G can, in general, be infinite the definition of $\succ$ is usually recursive. The step of showing $\Phi[v/t'] \Rightarrow \Phi[v/t]$, for every $t \succ t'$, is fragmented into several cases. Each of these cases is established using the relation $\rightarrow^*$ as was done in the equational logic. Fig. 12 gives an example of an inductive proof. It proves the property **Append(Append(q_1, q_2), q_3) ≡ Append(q_1, Append(q_2, q_3))** from the specification of **Queue_Int** given in Fig.11. The proof uses an ordering generated by the following relation on the generator expressions of **Queue_Int**: **Enqueue(q, i)** $\succ$ **Nullq**, and **Enqueue(q, i)** $\succ$ **q**. The proof uses the variable q_3 as the parameter of induction.

It is not possible to develop a general procedure to decide the entire inductive

---

Fig. 12. Proof by Inductive Logic

Theorem to be proved: **Append(Append(q_1, q_2), q_3) ≡ Append(q_1, Append(q_2, q_3))**

Basis: $q_3 \mapsto$ **Nullq**
To prove: **Append(Append(q_1, q_2), Nullq) ≡ Append(q_1, Append(q_2, Nullq))**

Proof is demonstrated above.

Induction: $q_3 \mapsto$ **Enqueue(q, si)**
Hypothesis: **Append(Append(q_1, q_2), q) → Append(q_1, Append(q_2, q))**
To prove: **Append(Append(q_1, q_2), Enqueue(q, i)) ≡ Append(q_1, Append(q_2, Enqueue(q, i)))**

Normal form of left hand side:	Normal form of right hand side:
Append(Append(q_1, q_2), Enqueue(q, i))	**Append(q_1, Append(q_2, Enqueue(q, i)))**
Rule(11) ↓	↓ **Rule(11)**
Enqueue(Append(Append(q_1, q_2), q), i)	**Append(q_1, Enqueue(Append(q_2, q), i))**
Hyp. ↓	↓ **Rule(11)**
Enqueue(Append(q_1, Append(q_2, q)), i)	**Enqueue(Append(q_1, Append(q_2, q)), i)**

theory of S. This is because the inductive hypotheses necessary for the proof cannot be generated automatically in all situations. However, when S satisfies the principle of definition a significant number of interesting properties in the inductive theory can be proved automatically. The automatic method, first developed by Musser [38, 22], is based on the Knuth-Bendix algorithm (see sec 3.3.3.1) for checking convergence of a rewriting system. We use this method for synthesizing implementations whose proofs of correctness need induction. We will explain the method in chapter 4 while describing synthesis in the inductive theory.

4. Stage 1: The Preliminary Implementation

This chapter discusses the preliminary implementation of a data type, and develops a method to derive it from the inputs to the synthesis procedure. A distinguishing characteristic of the method outlined is that it is based on a method for proving the correctness of a preliminary implementation. The chapter is organized into the following sections. The first section defines precisely what constitutes a preliminary implementation. The second section gives a mathematical formulation of the problem involved in the derivation of a preliminary implementation for a data type from the given inputs. For convenience, the problem is formulated, and solved here for a situation where the representing domain is identical to the representation value set. In the next chapter, we extend the derivation problem to the more general situation where the representing domain is a subset of the representation value set. The last section describes a procedure to derive the preliminary implementation from the input specifications.

4.1 A Preliminary Implementation

A preliminary implementation of a data type is an implementation for the implemented type in a rewrite rule language. The preliminary implementation uses a representation scheme that is consistent with the one characterized by the association specification supplied by the user. It consists of two parts: The *Representation* part, and the *Definitions* part.

The *Representation* part gives the representation type used for the implementation of the implemented type. We call the values of the representation type the *representation values*, and the set of representation values the *representation value set.* Only a subset of the representation value set need be used to represent the values of the implemented type. This subset is called the representing domain, and is characterized by the association specification.

The *Definitions* part contains definitions for a set of new functions on the representation values. We call the new functions the *implementing functions.* There is an implementing function corresponding to every operation of the implemented type; the former implements the latter. The definition of an implementing function that implements

an operation is called the *preliminary implementation* of that operation. An implementing function is not necessarily a total function on the representation value set. However, it has to be defined on every value of the representing domain. We use the following convention throughout the development to help associate an implementing function with the operation of the implemented type it implements: The identifier that denotes an implementing function is the capitalized version of the identifier that denotes the corresponding abstract operation. For instance, **NULLQ** is the implementing function of the operation **Nullq**.

The *Definitions* part consists of a set of rewrite rules of the form $\mathbf{e_1} \rightarrow \mathbf{e_2}$. The rewrite rules in the *Definitions* part defining an implementing function F are the ones that have F as the outermost symbol on their left hand side. $\mathbf{e_1}$ and $\mathbf{e_2}$ are expressions that may contain the implementing functions, the operations of the implementing types, and **if_then_else** with the following constraints:

(1) The only operations of the representation type that may appear in $\mathbf{e_1}$ and $\mathbf{e_2}$ are the generators of the type.

(2) $\mathbf{e_1}$ and $\mathbf{e_2}$ may not contain any auxiliary (or helping) functions other than **if_then_else**.

There are two reasons for constraining the preliminary implementation. Firstly, we would like to constrain the structure of the preliminary implementation so that the synthesis procedure has to perform less work in searching for the desired solution. Secondly, we want to keep the language as simple as possible so that the principle behind the synthesis method is brought out more clearly in our description.

The first constraint is imposed to keep the preliminary implementation derivation problem simple. This constraint permits us to ignore several axioms in the specifications of the implementing types during verification as well as synthesis of a preliminary implementation. In particular, the only axioms in the specification of the representation type that we need to consider are the ones that involve only the generators of the type involved in the specification. This is because only the generators of the representation type may appear in the preliminary implementation. To this extent this constraint simplifies the synthesis method. An implementation that also uses the rest of the operations is derived in the next

stage of the synthesis as a transformation of the preliminary implementation.

The second constraint, in general, restricts the logical power, i.e., the ability to define any computable function on the representation type, of the preliminary implementation language because the constraint prohibits the use of any helping (or auxiliary) functions (except **if_then_else**) in a preliminary implementation. Our synthesis method cannot automatically discover the helping functions that might be necessary in the preliminary implementation. We use two approaches to get around this problem; both the approaches amount to relaxing the second constraint. They are explained here briefly, but are illustrated more clearly when we later consider examples involving them.

The first approach consists of seeking help from the user. We require the user to furnish a specification of the helping function needed in the preliminary implementation. We then relax the second constraint to permit the use of the helping function in the preliminary implementation.

The second approach consists of introducing a new construct into the preliminary implementation language. The construct, which is used primarily in conjunction with a tuple type, helps eliminate the need for helping functions while defining several functions on tuple types. The motivation for paying special attention to tuple type is because a tuple type is a commonly used representation type. The construct provides a way of accessing the components of a tuple being returned by an expression of tuple type without explicitly using the operations that select the components of a tuple. This construct may be used in expressions that appear on the right hand side of an equation of a preliminary implementation. The construct is expressed by means of an expression with the following syntax:

e_2 where $\langle v_1, \ldots, v_n \rangle$ is e_{22}

In the above, $v_1, \ldots, v_n$ are variables; e_{22} is an expression of n-tuple type; e_2 is an expression that may contain the variables $v_1, \ldots, v_n$. The construct binds, in order, $v_1, \ldots, v_n$ to the components returned by e_{22}. The scope of the binding is limited to the expression e_2. For example, consider the expression

<Assign(v1, e, j1), i1, j1 + 1> where <v1, i1, j1> is DEQUEUE(<v, i, j>). Assuming **DEQUEUE** is a function from **Triple** to **Triple**, the variables v_1, i_1, and j_1 in the above

expression are bound to the components of the triple returned by **DEQUEUE(<v, i, j>)**.

4.2 The Preliminary Imlementation Derivation Problem

Our intention is to study the problem of synthesis within the data type verification framework. So we formulate the problem of deriving a preliminary implementation as roughly the inverse of the problem of proving the correctness of the preliminary implementation.

First, we develop the criterion of correctness of a preliminary implementation. Then, we formulate the problem of verifying if a preliminary implementation meets the correctness criterion. We define the derivation problem after that. For convenience, the verification problem and the derivation problem are formulated here for a situation in which the representing domain is identical to the representation value set. This situation corresponds to the case where the abstraction function is total, and the invariant part of the association specification is vacuous. We discuss the derivation problem for a situation where the representing domain is a subset of the representation value later. It should be noted that the formulation of the correctness criterion given below applies to both situations.

4.2.1 The Criterion of Correctness

Informally, for a preliminary implementation to be correct, the implementing functions it defines should collectively exhibit a behavior that is consistent with the observable behavior characterized by the specification of the implemented type. Also, the preliminary implementation should use a representation scheme that meets the requirements of the association specification given as input. Let us formalize this intuitive notion.

The formal object that a preliminary implementation is denoting can be considered to be a heterogeneous algebra, called the *implementation algebra*, with the following components:

(i) A principal domain that is a subset of the representation value set. The principal domain is defined as the set of all values of the representation type that are "reachable" through the implementing functions corresponding to the constructors

of the implemented type. In other words, the principal domain is the set of representation values generated by the closure under functional composition of the implementing functions corresponding to the constructors of the implemented type.

(ii) A domain corresponding to every defining type of the implemented type. We assume that this domain is identical to the value set of the corresponding defining type.

(iii) a function corresponding to every implementing function defined by the preliminary implementation.

A preliminary implementation is correct if the implementation algebra it denotes is a model of the implemented data type in a manner constrained by the association specification. This means that there exists a homomorphism from the implementation algebra to the the implemented type that behaves as an identity function on the values of the defining types, and exactly like the abstraction function characterized by the association specification on the values of the representation type.

Let $\mathfrak{R}$ denote the representing domain, and $\mathcal{A}$ denote the abstraction function specified by the association specification. Let $\mathcal{H}$ be a function defined as below.

D: Implemented Type, $\mathfrak{R}$: Representing Domain, $D_1, \ldots, D_n$:The defining types of **D**

$\mathcal{H}: \mathfrak{R} \cup D_1 \cup \ldots \cup D_n \rightarrow D \cup D_1 \cup \ldots \cup D_n$

$\mathcal{A}: \mathfrak{R} \rightarrow D$

$$\mathcal{H}(r) = \begin{cases} \mathcal{A}(r) & \text{if } r \in \mathfrak{R} \\ r & \text{otherwise} \end{cases}$$

A preliminary implementation of a data type is correct with respect to the association specification $\mathcal{A}$, if the following two conditions hold.

(1) *Totality Property*:Every implementing function is total over $\mathfrak{R}$.

(2) *Homomorphism Property*: The operation **f** of the implemented type and the implementing function **F** are related by the property: $(\forall r \in \mathfrak{R})[\mathcal{H}(F(..., r, ...)) = f(..., \mathcal{H}(r), ...)]$

The correctness criterion formulated above is different from the formulation found in the literature on data type verification [25, 14, 18] which is not formulated with respect to a given homomorphism $\mathfrak{H}$. According to the conventional formulation a preliminary implementation is correct if there exists a function $\mathfrak{H}$ from the representation value set to the value set of the implemented type so that: For all $r \in$ the principal domain, $\mathfrak{H}(F(..., r, ...)) = f(..., \mathfrak{H}(r), ...)$. Thus, according to this criterion the implementing functions are not required to be total with respect to $\mathfrak{R}$. Note that the principal domain can be a subset of $\mathfrak{R}$. What distinguishes our formulation is the requirement that F be total over $\mathfrak{R}$, and also satisfy the homomorphism property over $\mathfrak{R}$.

Our formulation is more useful in the context of synthesis. It enables us to determine a principal domain of the implementation algebra (which, in turn, determines the set of representation values on which every implementing function should be defined) directly from the association specification. This reduces the interdependence of the synthesis of preliminary implementation for the various operations of the type. This is because in other formulations the principal domain has to be determined by computing the closure under composition of the implementing functions of the constructors. Thus the domain of the implementing function of each of the constructors is, in general, dependent on the behavior of the implementing function of every other constructor.

The totality requirement is also more interesting in the context of synthesis. In the synthesis process the association specification initiates the derivation of an implementation by defining the representation scheme to be used. The association specification is expected to express the intention of the user regarding the representation scheme he wants the implementation (to be derived) to use. So it is logical to assume that the user wants the entire representing domain characterized by the association specification to be used for representing the values of the implemented type.

4.2.2 The Derivation Problem

The goal of the derivation problem is to derive a preliminary implementation from the given inputs so that the preliminary implementation meets the correctness criterion. The inputs consist of the specification of the implemented type, the specification of the implementing types, and the *homomorphism specification.* The homomorphism specification is a specification of the homomorphism $\mathfrak{H}$ that the preliminary implementation ought to obey. This specification is easily derived from the specification of the abstraction function $\mathcal{A}$ (given as a part of the association specification). The Homomorphism Specification contains two kinds of rewrite rules obtained as described below. The first set of rules specifies that $\mathfrak{H}$ behaves exactly like the abstraction function on the representation values. The second set of rules specifies that $\mathfrak{H}$ behaves as an identity function on the values of all the ancillary types. More precisely,

(1) if $\mathcal{A}(e_1) \equiv e_2$ belongs to the abstraction function specification

then $\mathfrak{H}(e_1) \equiv e_2$ belongs to Homomorphism Specification

(2) if σ is a generator of an ancillary type

then $\mathfrak{H}(\sigma(v_1, \ldots, v_n)) \equiv \sigma(\mathfrak{H}(v_1), \ldots, \mathfrak{H}(v_n))$ belongs to Homomorphism Specification

Let us call the combination of all the input specifications the *Input World* (**IW**). The restrictions on the inputs (see sec 2.3.1 of the previous chapter) ensure that the Input World satisfies the principle of definition. The strategy behind the method used in deriving the preliminary implementation is based on the principle of definition property.

Suppose **IW** is supplemented with a set of rewrite rules, called the $\mathfrak{H}$-rules, that express the homomorphism property a preliminary implementation is expected to satisfy: For every pair of an operation **f** of the implemented type, and its implementing function **F** there exists an $\mathfrak{H}$-rule of the form $\mathfrak{H}(F(v_1, \ldots, v_n)) \rightarrow f(\mathfrak{H}(v_1), \ldots, \mathfrak{H}(v_n))$. Let us call the supplemented system the *Perturbed World* (**PW**). Let us suppose that the addition of the $\mathfrak{H}$-rules does not destroy the uniform termination property of **IW**. The reason we refer to the supplemented system as the Perturbed World is because the addition of the $\mathfrak{H}$-rules destroys the principle of definition property. **PW** does not satisfy the principle of definition because the implementing functions that are newly introduced into the system are as yet undefined.

A constant involving the implementing function symbols does not simplify to a generator constant.

Recall that the principle of definition is a formal expression of the requirement that every nongenerator function in a system be completely defined as a total function. If we can generate a set of rewrite rules that can restore the principle of definition property of **PW**, then the new set of rules can be considered as a complete definition for the implementing functions. Thus, preliminary implementation derivation is a problem of restoring the principle of definition of a system that violates it.

More precisely, the problem involved in synthesizing a preliminary implementation consists of deriving from the Perturbed World a set of rewrite rules, **PI** (the acronym stands for preliminary implementation), so that

(1) **PI** ∪ **IW** satisfies the principle of definition, as well as

(2) **PI** ∪ **PW** satisfies the principle of definition.

In the following, we give a formal proof that the above conditions guarantee the correctness of the preliminary implementation.

The Correctness Theorem

Let **PI** be a set of rewrite rules derived so that the above two conditions hold. Then, **PI** satisfies the criterion of correctness of a preliminary implementation.

Proof The first condition asserts that **PI** ∪ **IW** satisfies the principle of definition. This implies that every nongenerator function in the system, which includes every implementing function, is defined as a total function. Hence, **PI** satisfies the Totality Property.

To show that **PI** satisfies the Homomorphism Property, we have to show that every equation of the form $\mathcal{H}(F(v_1, \ldots, v_n)) \equiv f(\mathcal{H}(v_1), \ldots, \mathcal{H}(v_n))$ is a theorem of **PI** ∪ **IW**. The argument to show that the second condition implies this is based on the following interesting

result about any system that satisfies the principle of definition. The result,[14] which is proved as **Theorem** 6 in Appendix III, enunciates a sufficient condition for an equation to be a theorem of a system that satisfies the principle of definition. Suppose S is a system that satisfies the principle of definition, and $e_1 \equiv e_2$ is an equation so that e_1 and e_2 have at least one nongenerator function symbol in them. Then, $e_1 \equiv e_2$ is a theorem of S if $S \cup \{e_1 \rightarrow e_2\}$ satisfies the principle of definition. The result is proved in the **Lemma** to follow.

Because of the second condition **PI** ∪ **PW** satisfies the principle of definition. Since **PW** is **IW** ∪ $\mathfrak{H}$-rules, this implies that (**PI** ∪ **IW**) ∪ $\mathfrak{H}$-rules satisfies the principle of definition. Now, by the first condition **PI** ∪ **IW** satisfies the principle of definition. By applying the above result, each of the $\mathfrak{H}$-rules (when treated as equations) is a theorem of **PI** ∪ **IW**. Note that the result can be applied because the $\mathfrak{H}$-rules have nongenerator function symbols in them.

Q.E.D.

4.3 Derivation of a Preliminary Implementation

In the previous section the problem of deriving a preliminary implementation was formulated as deriving a set of rewrite rules, **PI**, so as to restore the principle of definition property to the Perturbed World **PW**. This section develops a procedure to derive a preliminary implementation. The procedure makes two assumptions about its input: (1) The Initial World (**IW**) satisfies SCPD, a sufficient condition for the principle of definition, and (2) a termination ordering $\succ$ on expressions is available to the procedure to ensure the uniform termination property of rewriting systems.

The obvious strategy for the procedure is to derive the rules of the preliminary implementation so that **PI** ∪ **IW** and **PI** ∪ **PW** satisfy SCPD. But this limits the class of

---

14. [22, 38] contain results similar to the one proved in this lemma. The result here is different because we have a different set of assumptions. The principle of definition property used in [22] is more constrained than the one we have. The result in [38] assumes that S satisfies a completeness property called *fully specifiedness* which is not assumed here. This is the reason for the requirement in the lemma that e_1 and e_2 should have at least one nongenerator function symbol in it.

implementations that can be derived by the procedure. So, we develop another set of conditions, called the *synthesis conditions*, that is weaker than SCPD. **PI** is generated so that it satisfies the synthesis conditions. It can be shown that when **PI** satisfies the synthesis conditions, **PI** ∪ **IW** and **PI** ∪ **PW** satisfy the principle of definition. We first formulate the synthesis conditions, and then develop a procedure to derive a set of rules that satisfies the synthesis conditions.

4.3.1 The Synthesis Conditions

The synthesis conditions for a set of rewrite rules **PI** are the following:

(1) **Totality Condition:**

(a) **PI** is well-spanned (for every implementing function) with every rule in it being of the form $\mathbf{F(g_1, \ldots, g_n) \rightarrow t}$,[15] where **F** is an implementing function symbol, and $\mathbf{g_1, \ldots, g_n}$ are generator expressions.

(b) **PI** satisfies the uniform termination property.

(2) **Uniqueness Condition:** **PI** has the unique termination property.

(3) **Homomorphism Condition:** For every rule $\mathbf{F(g_1, \ldots, g_n) \rightarrow t}$ in **PI**, $\mathfrak{H}(\mathbf{F(g_1, \ldots, g_n)}) \equiv \mathfrak{H}(\mathbf{t})$ is a theorem of **PW**.

The following **Synthesis Theorem** shows that when **PI** satisfies the synthesis conditions, **PI** ∪ **IW** and **PI** ∪ **PW** satisfy the principle of definition, and hence, by the **Correctness Theorem**, **PI** is correct. An informal motivation for the conditions can be given as follows. The Totality Condition ensures that every implementing function is defined on all the values of the representation type, and it terminates on each of them. The Uniqueness Condition ensures that every implementing function is well-defined, in the sense that it yields a unique value for every argument value. The Homomorphism Condition ensures that the preliminary

---

15. Note that the syntactic constraint on a preliminary implementation requires that **t** may contain neither the function symbol $\mathfrak{H}$, nor any of the operations of the implemented type.

implementation satisfies the Homomorphism Property.

The Synthesis Theorem

If **PI** satisfies the synthesis conditions, then **PI** ∪ **IW** and **PI** ∪ **PW** satisfy the principle of definition, and hence **PI** is a correct preliminary implementation.

Proof It is easy to see that **PI** ∪ **IW** satisfies the principle of definition because the Totality Condition and the Uniqueness Condition imply that preliminary implementation satisfies SCPD, and **IW** satisfies SCPD by our assumption about the inputs.

Let **NW** denote **PI** ∪ **PW**, for convenience. We apply Theorem 8 (Appendix III) to show that **NW** satisfies the principle of definition. According to that theorem, a rewriting system S satisfies the principle of definition if

(a) S is well-spanned,

(b) S has the uniform termination property

(c) Every critical pair $\langle \alpha_1, \alpha_2 \rangle$ of S is such that $\alpha_1 \equiv \alpha_2$ is a theorem of S.

We show that **NW** satisfies all three premises of the above theorem. **NW** is well-spanned. This is because **IW** is well-spanned by our assumption, and **PI** is well-spanned by Totality Condition (a). The only nongenerator function symbols of **NW** are the ones in **IW** and **PI**. By Totality Condition (b) **PI** has the uniform termination property, so **NW** has the uniform termination property also. The following lemma shows that **NW** satisfies premise (c).

Q.E.D.

Lemma Every critical pair $\langle \mathbf{e}_1, \mathbf{e}_2 \rangle$ of **NW** is such that $\mathbf{e}_1 \equiv \mathbf{e}_2$ is a theorem of **NW**.

Proof Note that **PW** is convergent. This is because **IW** is convergent by assumption, and the $\mathfrak{R}$-rules added to **IW** do not give rise to any new critical pairs.

NW is constructed from **PW** by adding **PI** to the former. Therefore, any new critical pairs of **NW** would be generated as a result of a superposition of the rules of **PI** on the rules of **NW**. Because of Totality Condition (a) on the form of the rules in **PI** the only rules

on which the rules of PI can have a superposition are the following:

(I) The rules of PI themselves, or

(II) the rules of the implementing types,

(III) the $\mathfrak{H}$-rules.

Every critical pair $\langle e_1, e_2 \rangle$ determined by a superposition on the rules in category (I), and (II) is such that $e_1\downarrow$ is identical to $e_2\downarrow$. This is because, by the Uniqueness Condition, PI satisfies the unique termination property. Hence, $e_1 \equiv e_2$ is a theorem of NW.

Every critical pair determined by a superposition of the rules in category (III) is of the form $\langle \mathfrak{H}(F(g_1, \ldots, g_n)), \mathfrak{H}(t) \rangle$, where $F(g_1, \ldots, g_n) \rightarrow t$ is a rule in PI. By the Homomorphism Condition, $\mathfrak{H}(F(g_1, \ldots, g_n)) \equiv \mathfrak{H}(t)$ is a theorem of PW, and hence a theorem of NW.

Q.E.D.

4.3.2 Derivation of the Rules of PI

The rewrite rules of PI are derived from the Perturbed World (PW). So the initial task of the derivation procedure is to construct PW. PW is a rewriting system that includes the Initial World (IW) and the $\mathfrak{H}$-rules. IW is constructed by combining the specification of the implemented type, the specifications of the implementing types, and the Homomorphism Specification. Without any loss of generality, we assume that there is no conflict among the names of the various function symbols in the specifications. PW is formed by then adding a rule of the form $\mathfrak{H}(F(v_1, \ldots, v_n)) \rightarrow f(\mathfrak{H}(v_1), \ldots, \mathfrak{H}(v_n))$ for every implementing function F to be defined. We assume that the termination ordering $\succ$ being used by the synthesis procedure is such that $\mathfrak{H}(F(v_1, \ldots, v_n)) \succ f(\mathfrak{H}(v_1), \ldots, \mathfrak{H}(v_n))$, for every implementing function. This ensures that PW retains the uniform termination property as desired by the derivation problem. Note that this is not a restriction because the implementing function symbols (in the $\mathfrak{H}$-rules) are fresh symbols being introduced into IW. Hence, an appropriate ordering can always be found.

Although PW is defined to include the specification of every implementing type

completely, it is not necessary to do so. Since the derivation method does not require the specifications to be complete, one may include only parts of the specifications of the implementing types. The advantage of doing so is that the fewer rules in **PW** the more efficient it is to derive the preliminary implementation. However, by not including certain rewrite rules one might be excluding certain implementations.

Let us illustrate the construction of **PW** on an example. We consider the derivation of an implementation for **Queue_Int** with **Circ_List** as the representation type using the association specification given in Fig. 9 in the previous chapter. Fig. 13 gives the rules of **PW** for the example under consideration. The rules of the types **Integer** and **Bool**, which are also among the implementing types are omitted from the figure for convenience. The rules of the

---

Fig. 13. The Perturbed World

(1) Front(Nullq) → ERROR
(2) Front(Enqueue(Nullq, e)) → e
(3) Front(Enqueue(Enqueue(q, e1), e2)) → Front(Enqueue(q, e1))

(4) Dequeue(Nullq) → ERROR
(5) Dequeue(Enqueue(Nullq, e)) → Nullq
(6) Dequeue(Enqueue(Enqueue(q, e1), e2)) → Enqueue(Dequeue(Enqueue(q, e1)), e2)

(10) Append(q, Nullq) → q
(11) Append(q1, Enqueue(q2, e2)) → Enqueue(Append(q1, q2), e2)

(12) Empty(Nullq) → True
(13) Empty(Enqueue(q, e)) → False

(14) ℋ(Create) → Nullq
(15) ℋ(Insert(c, i)) → add_at_head(ℋ(c), ℋ(i))

(16) add_at_head(Nullq, i) → Enqueue(Nullq, i)
(17) add_at_head(Enqueue(q, i), i1) → Enqueue(add_at_head(q, i1), i)

(19) ℋ(NULLQ()) → Nullq
(20) ℋ(ENQUEUE(c, i)) → Enqueue(ℋ(c), ℋ(i))
(21) ℋ(DEQUEUE(c)) → Dequeue(ℋ(c))
(22) ℋ(APPEND(c1, c2)) → Append(ℋ(c1), ℋ(c2))
(23) ℋ(EMPTY(c)) → Empty(ℋ(c))

representation type Circ_List are omitted because they are not going to be used in the derivation of the preliminary implementation. This situation arises because a preliminary implementation is permitted to use only the generators of the representation type. So, the only rules of the representation type needed in verification, and hence also in the derivation of a preliminary implementation, are the ones that contain only the generators. Since Circ_List does not have any rules of this kind, Circ_List does not contribute any rules to IW. Rules (1) through (13) in the figure are rules of Queue_Int; rules (14) through (17) are the rules of Homomorphism Specification.

The next task is to derive the rewrite rules of PI from PW. Strictly speaking, PI should be derived so that all the three synthesis conditions are satisfied. But, it is more convenient to develop a procedure that derives the rewrite rules so that only the Totality Condition and the Homomorphism Condition are met. The effect of ignoring the Uniqueness Condition is not harmful in the sense that it can be fixed at a later stage by post-processing the preliminary implementation. The Uniqueness Condition ensures that every implementing function defined by PI returns a unique value on every representation value. When the Uniqueness Condition is not satisfied, an implementing function F being defined by PI may be nondeterministic: That is, F can be so that $F(v) = v_1$, and $F(v) = v_2$, but $v_1 \neq v_2$; however, both the values v_1 and v_2 will represent the same value of the implemented type. The nondeterministic behavior, if any, in the preliminary implementation will be eliminated by our synthesis procedure in the second stage while deriving a target implementation. The semantics of the target implementation language is such that it is impossible to define nondeterministic functions.

The procedure derives the preliminary implementation for one operation at a time by deriving a separate set of rewrite rules for every operation. The method used is the same for every operation. The procedure first determines the left hand sides of all the rules of the preliminary implementation. Then, it determines a suitable right hand side for each of the rules from the already determined left hand side.

4.3.2.1 Determining the Left Hand Side

The Totality Condition is used to determine the left hand side of the rules. The Totality Condition has two parts: The first part requires PI to be well-spanned, and the second part requires PI to have the uniform termination property. The second part is ensured while deriving the right hand side, which will be discussed later. The first part is used here.

The well-spannedness property (described formally in sec 2.3.1 of the previous chapter) requires the left hand side expressions of the rules defining an implementing function F to satisfy the following property: The set of generator expressions the appear as arguments to F on the left hand side should span the set of all generator constants. More precisely, suppose the preliminary implementation of F consists of the following set of rules: (In the following the question mark identifiers are used as place holders for expressions to be determined later.)

$F(g_1) \rightarrow ?t_1$

.....

$F(g_n) \rightarrow ?t_n$

Then, the set $\{g_1, \ldots, g_n\}$ should be well-spanned (see sec 2.3.1), i.e., span the set of all generator constants of the appropriate implementing type. For instance, as a concrete example, any pair of rules that have the form given below constitute a well-spanned set of rules for **ENQUEUE**.

ENQUEUE(Create, j) $\rightarrow$ **?rhs**$_2$

ENQUEUE(Insert(c, i), j) $\rightarrow$ **?rhs**$_3$

Note that the left hand side of each of the above rules consists of **ENQUEUE** applied to arguments that are generator expressions. The set of arguments, i.e., sequences of generator expressions, to **ENQUEUE** on the left hand side of the rules is **ArgsSet** = {<**Create, j**>, <**Insert(c, i), j**>}. **ArgsSet** spans the set of all ordered pairs of generator constants because every pair of generator constants (the first one of type **Circ_List**, and the second of type **Integer**) is an instance of one of the arguments in **ArgsSet**.

It is easy to build a procedure that automatically generates a well-spanned **ArgsSet**,

once the generators of the representation type are identified. In fact a slight modification to the procedure referred in sec 3.3.3 (which checks if an **ArgsSet** is complete) can be used to generate a complete set of argument expressions. Thus, an appropriate set of left hand sides for the rewrite rules to be derived can be determined automatically.

Fig. 14 gives a possible set of left hand side expressions for a preliminary implementation for the example under consideration. Note that the right hand side of each of the rules in the figure is denoted by a question mark identifier. So Fig. 14 can be considered as a partial preliminary implementation of **Queue_Int.**

4.3.2.2 Determining the Right Hand Side

The right hand side of each of the rules is determined using the already determined left hand side so that the Homomorphism Condition and the second part of the Totality Condition are met. This where the Perturbed World (**PW**) comes into the picture.

PW is used to derive a set of equations, called the *synthesis equations*, one equation for every rule in the preliminary implementation. The right hand side of a rule is determined from the right hand side of the corresponding synthesis equation. The synthesis equation

---

Fig. 14. A Partial Preliminary Implementation

(1) NULLQ() → ?rhs$_1$

(2) ENQUEUE(Create, j) → ?rhs$_2$
(3) ENQUEUE(Insert(c, i), j) → ?rhs$_3$

(4) FRONT(Create) → ?rhs$_4$
(5) FRONT(Insert(c, i)) → ?rhs$_5$

(6) DEQUEUE(Create) → ?rhs$_6$
(7) DEQUEUE(Insert(c,i)) → ?rhs$_7$

(8) APPEND(c, Create) → ?rhs$_8$
(9) APPEND(c, Insert(d, i)) → ?rhs$_9$

(10) SIZE(Create) → ?rhs$_{10}$
(11) SIZE(Insert(c, i)) → ?rhs$_{11}$

corresponding to a rewrite rule $F(g_1) \rightarrow ?t_1$ is an equation of the form $\mathcal{H}(F(g_1) \equiv \mathcal{H}(?t_1)$ that satisfies the following conditions:

(1) $\mathcal{H}(F(g_1) \equiv \mathcal{H}(?t_1)$ is a theorem of **PW**

(2) $\mathcal{H}(F(g_1) \succ \mathcal{H}(?t_1)$, where $\succ$ is the termination ordering on expressions.

(3) $?t_1$ contains the implementing function symbols and the permitted operations of the implementing types.

it is easy to see the justification for the above conditions. The first condition contributes towards ensuring the Homomorphism Condition. The second condition ensures the uniform termination property. The third condition is just a syntactic constraint that any rule in a preliminary implementation ought to satisfy. The next section describes in detail a procedure to derive the synthesis equations.

4.4 Deriving the Synthesis Equations

Every synthesis equation of the preliminary implementation is derived with the help of two inference rules called the *synthesis rules*. The synthesis rules are designed for generating theorems of **PW** that have the same left hand sides, but different right hand sides. For deriving a synthesis equation, the synthesis rules are invoked repeatedly a finite number of times to generate a series of theorems until the desired equation is generated. For instance, the synthesis equation corresponding to the rule **ENQUEUE(Insert(c, i), j)** $\rightarrow$ $?rhs_2$ (in the partially derived preliminary implementation given in Fig. 14) is derived by generating a series of theorems that have $\mathcal{H}$**(ENQUEUE(Insert(c, i), j))** as their left hand side. The generation continues until a theorem whose right hand side qualifies the theorem to be a synthesis equation is encountered.

We investigate two ways in which the synthesis rules can be used for deriving a synthesis equation. The first one derives synthesis equations that are in the equational theory of **PW**. The second one derives equations that are in the inductive theory. The second method is more general than the first one. A system that implements the synthesis procedure would, therefore, use only the second method. We discuss them separately for pedagogic

reasons. First, we formulate the synthesis rules. The subsequent subsections describe the use of the synthesis rules in deriving the synthesis equations.

4.4.1 The Synthesis Rules

The idea used for generating an equation is to reverse the method of demonstrating that the equation is a theorem of PW. The central notion used in the generation is *expansion*. Expansion is the opposite of reduction. It is the act of applying a rewrite rule to an expression from right to left.

4.4.1.1 Informal Explanation

The basis for the synthesis rules is the result given in the **KB-Theorem** (sec 3.3.3.1). The theorem gives rise to the following principle for generating equations that are theorems of a convergent system. Suppose e_1 is an expression that we wish to have as the left hand side of the equation. Then, an expression $?e_2$ that may appear on the right hand side of any equation that has e_1 as its left hand side should be such that $e_1{\downarrow} = ?e_2{\downarrow}$. One way of ensuring that $?e_2$ simplifies to $e_1{\downarrow}$ is to obtain $?e_2$ by applying to $e_1{\downarrow}$ the rewrite rules of the system from right to left a finite number of times. We call the mechanism of applying a rule to an expression from right to left *expand*.

We will give a formal definition of expand, and discuss its properties later. Here, we will give an approximate description of what expand does so that we may develop a first version of the synthesis rule, and illustrate them on the example.[16] Like reduce, performing expand consists of several steps. Suppose we wish to expand **Add_at_head(Enqueue(ℑ6(c), ℑ6(j)), ℑ6(i))** using the rule **ℑ6(ENQUEUE(c, j)) → Enqueue(ℑ6(c), ℑ6(j))**. One way of doing this is to look for a subexpression (inside the expression to be expanded) that has the form of the right hand side

---

16. We will generalize the definition of expand later. At that point one of the synthesis rules needs to revised slightly as well. According to the definition given here, expansion is identical to the transformation technique *folding* used by Darlington [7] for synthesis of recursive programs.

of the rule. Then replace the subexpression by the corresponding instance of the left hand side of the rule. In the present case, the subexpression that appears as the first argument to **Add_at_head** in the given expression matches the right hand side of the rule for the identity substitution. The result of expanding the expression is then **Add_at_head(ℋ(ENQUEUE(c, j)), ℋ(i))**. The result of expanding an expression **e** in the occurrence u by a rule $\gamma \rightarrow \delta$ is denoted by *expand* **e** *in* u *by* $\gamma \rightarrow \delta$. We use **expand(e)** to denote any expression that is obtained by expanding **e** in some occurrence u by some rule $\gamma \rightarrow \delta$ in the rewriting system under consideration.

We are now in a position to give the synthesis rules. The first rule specifies how to start the generation of a series of theorems; it generates a theorem from a given expression without the need for any existing theorem.

Rule 1: $$\frac{\text{e is an expression}}{\mathbf{e \equiv e\downarrow}}$$

The second rule specifies a way of generating a new theorem from an existing one using expand.

Rule 2: $$\frac{\mathbf{e_1 \equiv e_2}}{\mathbf{e_1 \equiv expand(e_2)}}$$

To familiarize the reader with the synthesis rules let us invoke each of the synthesis rules to generate a couple of theorems that have **ℋ(ENQUEUE(Insert(c, i), j))** as their left hand. We use the rewrite rules of **PW** given in Fig.pw1 for expansion and reduction. The normal form of **ℋ(ENQUEUE(Insert(c, i), j))** is **Enqueue(Add_at_head(ℋ(c), ℋ(i)), ℋ(j))**, which is obtained by using the rewrite rule (20) and then (15) for simplification. By invoking synthesis rule (1) with **e** = **ℋ(ENQUEUE(Insert(c, i), j))**, we generate the following theorem of **PW**:

ℋ(ENQUEUE(Insert(c, i), j) ≡ Enqueue(Add_at_head(ℋ(c), ℋ(i)), ℋ(j))

Let us now invoke synthesis rule (2) on the above equation. Using the rewrite rule (17) to expand the entire expression on the right hand side of the above theorem, we can generate the following theorem of **PW**:

ℋ(ENQUEUE(Insert(c, i), j) ≡ Add_at_head(ℋ(ENQUEUE(c, j)), ℋ(i))

4.4.1.2 Formal Definition of Expand

Expansion is roughly the reverse of the process of reduction. The relation that characterizes a single step of expansion is called *expand.* Expanding an expression using a rule is close to applying the rule to the expression from right to left.

The motivation for introducing the mechanism of expansion is to solve a common problem encountered during synthesis: This is to find an expression (a *desired expression*) that simplifies to given expression (the *starting expression*). For instance, in the derivation shown earlier, the starting expression was **Enqueue(Add_at_head(ℑ(c), ℑ(i)), ℑ(j))**, and the desired expression was **ℑ(Insert(ENQUEUE(c, j), i))**.

The definition of expand uses the concept of unification, and the most general unifier (see Appendix I). Let **t** be an expression, and $\gamma \rightarrow \delta$ be a rule. We assume that **t** and γ have disjoint variable sets. If there are common variables then they have to be renamed suitably. Let u be an occurrence in **t** such that $\mathbf{t}/u$ is unifiable with δ; let θ be the most general unifier. Let $\mathbf{t}'$ be the expression $\mathbf{t}[u \leftarrow \theta(\gamma)]$. Then, we say that **t** *expands to* $\mathbf{t}'$ *by* $\gamma \rightarrow \delta$ *in* u; we denote this relation by $\mathbf{t} \leftarrow \mathbf{t}'$. Notice that expanding **t** by $\gamma \rightarrow \delta$ in u is not equivalent to reducing **t** by $\delta \rightarrow \gamma$ in u. Expand checks if $\mathbf{t}/u$ is unifiable with δ, whereas reduce checks if $\mathbf{t}/u$ has the form of δ. Therefore, there are situations where an expression is expandable by $\gamma \rightarrow \delta$, but not reducible by $\delta \rightarrow \gamma$.

The following question arises immediately: Why was expand not defined exactly as applying a rule in the reverse direction ? The reason is that a rule $\gamma \rightarrow \delta$ may be such that $\text{varset}(\gamma) \supset \text{varset}(\delta)$. Applying such a rule from right to left will result in an expression that contains "new" variables, i.e., variables that did not exist in the original expression. The use of such variable dropping rule during reduction represents a situation where the reduction step caused a "loss" of information: A new variable introduced in an expansion step might have had in its place an arbitrary expression during the corresponding reduction step. Our goal is to reconstruct, if possible, this lost information at a later stage in the expansion process. During expansion, therefore, a variable in an expression has to be treated, in general, as though an arbitrary expression might be in its place. Using the predicate *unifiable* to determine if an expression is expandable enables us to do this.

For instance, consider the expansion of **Append(q, Nullq)** by the rule **Dequeue(Enqueue(Nullq, e)) → Nullq**. The resulting expression is **Append(q, Dequeue(Enqueue(Nullq, e)))**. The variable **e** is a new variable introduced because of expansion. Every instance of the latter expression in which **e** is replaced by any other expression reduces to the former expression. It might be possible to determine the expression that has to take the place of **e** in future expansion steps.

It should be pointed out, however, that not all variables in an expression need be given such a special treatment during expansion. The variables that appear in the starting expression must appear as they are in the desired expression we are shooting for. Therefore, while expanding an expression, it is necessary to distinguish between the variables in the expression that were introduced by a rule (presumably during earlier steps of expansion) and the ones that were transferred to the expression from the starting expression. We classify the variables involved in expansion into the following two kinds:

(1) The variables appearing in the rewrite rules; we continue to call these *variables.*

(2) The variables appearing in the expressions on the left hand sides of the rewrite rules in the partially generated preliminary implementation (Fig. 14). We call these variables *terminals.* Henceforth, we denote terminals by identifiers that are in italics.

The definition of an expression remains as before except that it may also contain terminals in it. The definition of a substitution also remains as before; it is a function from variables to expressions. Thus, when a substitution is extended to be applicable on an expression, the terminals in the expression are not substituted for, as we desired.

In the wake of the formal definition of expand, and the preceding discussion about the introduction of variables into expressions due to expansion, we should reconsider the formulation of the synthesis rules. The first synthesis rule remains unchanged because it does not use the relation expand. The second synthesis rule was formulated as below:

Rule 2: $$\frac{e_1 \equiv e_2}{e_1 \equiv \mathbf{expand}(e_2)}$$

This formulation is not general enough because it does not account for all the theorems that can be derived from $e_1 \equiv e_2$ in one expansion step. If **expand**(e_2) has variables in it, then every instance of it can potentially be the right hand side of a theorem. Hence, we re-formulate the rule as follows:

Rule 2:
$$\frac{e_1 \equiv e_2,\ \sigma \text{ is a substitution}}{e_1 \equiv \sigma(\textbf{expand}(e_2))}$$

4.4.2 Derivation in the Equational Theory

As an illustration, let us derive a synthesis equation that is of the form $\mathcal{H}$(ENQUEUE(Insert(c, i), j)) $\equiv \mathcal{H}(?rhs_3)$ in the partial preliminary implementation shown in Fig. 14. The equation is derived by generating a series of theorems that have $\mathcal{H}$(ENQUEUE(Insert(c, i), j)) as their left hand side. The generation is begun by invoking synthesis rule (1) on the left hand side expression. The rest of the theorems in the series are generated by invoking synthesis rule (2) using the rewrite rules of PW for expansion. The rewrite rules for expansion are chosen with the following ultimate goal: Obtain a right hand side that has the form $\mathcal{H}(?rhs_3)$ so that $\mathcal{H}$(ENQUEUE(Insert(c, i), j)) $\succ \mathcal{H}(?rhs_3)$, and $?rhs_3$ contains only the permitted operations of the implementing types. In the illustration given below, the generation of every theorem in the series is considered as a step. At each step, the expression expanded, and the rewrite rule used for expansion are indicated.

Relevant Rewrite Rules of the Perturbed World

(1) $\mathcal{H}$(ENQUEUE(c, j)) → Enqueue($\mathcal{H}$(c), $\mathcal{H}$(j))

(2) $\mathcal{H}$(Create) → Nullq

(3) $\mathcal{H}$(Insert(c, i)) → Add_at_head($\mathcal{H}$(c), i)

(4) Add_at_head(Nullq, i) → Enqueue(Nullq, i)

(5) Add_at_head(Enqueue(q, i), j) → Enqueue(Add_at_head(q, j), i)

Form of the theorem to be generated: $\mathcal{H}$(ENQUEUE(Insert(c, i), j)) $\equiv \mathcal{H}(?rhs_3)$
Normal form of $\mathcal{H}$(ENQUEUE(Insert(c, i), j)): Enqueue(Add_at_head($\mathcal{H}$(c), i), $\mathcal{H}$(j))
Rules used for the normal form: (1), (3)

Step (1) Invoke Synthesis Rule (1) on $\mathcal{H}$(ENQUEUE(Insert(c, i), j))
$\mathcal{H}$(ENQUEUE(Insert(c, i), j)) $\equiv$ Enqueue(Add_at_head($\mathcal{H}$(c), i), $\mathcal{H}$(j))

Step (2) Expand Expression: **Enqueue(Add_at_head(ℑ(c), i), ℑ(j))**
Using Rule: (5)

ℑ(ENQUEUE(Insert(c, i), j)) ≡ Add_at_head(Enqueue(ℑ(c), ℑ(j)), i)

Step (3) Expand Expression: **Enqueue(ℑ(c), ℑ(j))**
Using Rule: (1)

ℑ(ENQUEUE(Insert(c, i), j)) ≡ Add_at_head(ℑ(ENQUEUE(c, j)), i)

Step (4) Expand Expression: **Add_at_head(ℑ(ENQUEUE(c, j)), i)**
Using Rule: (3)

ℑ(ENQUEUE(Insert(c, i), j)) ≡ ℑ(Insert(ENQUEUE(c, j), i))

The theorem generated in step (4) qualifies to be a synthesis equation.
Hence the desired rule of the preliminary implementation is:

ENQUEUE(Insert(c, i), j) → Insert(ENQUEUE(c, j), i)

4.4.3 Derivation in the Inductive Theory

4.4.3.1 The General Strategy

The method used for deriving a synthesis equation in the inductive theory is based on the following property that every theorem of **PW** satisfies: If an equation is a theorem of **PW**, then every instance of it is in the equational theory of **PW**. An instance of an equation $e_1 \equiv e_2$ is an equation obtained by replacing every variable in e_1 and e_2 by generator constants.

We, therefore, take the following approach. Suppose the synthesis equation we

wish to derive is of the form $\mathfrak{I}\mathfrak{G}(F(e_{11})) \equiv \mathfrak{I}\mathfrak{G}(?e_{12})$.[17] We first derive an instance of the desired equation: This is done by selecting an instance of the left hand side, say $\sigma(\mathfrak{I}\mathfrak{G}(F(e_{11})))$, for some substitution σ of the terminals in e_{11} to generator constants. Then, an instance of the equation $\sigma(\mathfrak{I}\mathfrak{G}(F(e_{11}))) \equiv \sigma(\mathfrak{I}\mathfrak{G}(e_{12}))$ is derived; the method of derivation for the equational theory described earlier can be used for this purpose. The instance of the equation derived should be such that a generalization of it $\mathfrak{I}\mathfrak{G}(F(e_{11})) \equiv \mathfrak{I}\mathfrak{G}(e_{12})$, which is obtained by replacing assorted constants by suitable terminals in the instance, is a theorem of PW.

To check if the generalization is a theorem of PW, we use an automatic procedure called **Is-an-inductive-theorem-of**. This procedure is capable of deciding a significant number of theorems in the inductive theory of a system. The procedure will be described in a subsequent subsection. Another topic that will be deferred until later is determining a suitable σ. Any substitution that maps all the terminals in the left hand side of the synthesis equation to arbitrary generator constants will serve our purpose. However, the derivation would be more efficient if we instantiated as few terminals as possible. A later subsection will discuss a method of determining a more judicious way of choosing σ.

In the rest of this subsection, we formalize the notion of the generalization of an equation, and then illustrate the general strategy by deriving a synthesis equation corresponding to the rewrite rule **APPEND(*c*, Insert(*d*,*i*))** $\rightarrow$ **?rhs$_9$** in the partial preliminary implementation of **APPEND** given in Fig. 14.

The Generalization of an Equation

The *generalization* of an equation $e_1 \equiv e_2$ with respect to a substitution σ is the set of equations such that $e_1 \equiv e_2$ is an instance of using σ. When the substitution with respect to which the equation is being generalized is obvious from the context, we denote the generalization by **Gen**$[e_1 \equiv e_2]$. Formally, every equation $e_1' \equiv e_2' \in$ **Gen**$[e_1 \equiv e_2]$ is such that $\sigma(e_1') = e_1$, and $\sigma(e_2') = e_2$. Note that if $e_1 \equiv e_2$ has a finite number of function symbols **Gen**$[e_1 \equiv e_2]$ is always finite. For instance, suppose σ is $\{d \mapsto$ **Create**$\}$.

---

17. Recall that the left hand side of the synthesis equation is already known.

Then, **Gen[$\mathcal{H}$(Append(*c*, Insert(Create, i))) ≡ $\mathcal{H}$({APPEND(ENQUEUE(*c*, *i*), Create))})]** contains the following equations:

$\mathcal{H}$(Append(*c*, Insert(Create, i))) ≡ $\mathcal{H}$({APPEND(ENQUEUE(*c*, *i*), Create))})

$\mathcal{H}$(Append(*c*, Insert(*d*, i))) ≡ $\mathcal{H}$(APPEND(ENQUEUE(*c*, *i*), *d*))

As an illustration let us derive an equation of the form **$\mathcal{H}$(APPEND(*c*, Insert(*d*,*i*))) ≡ $\mathcal{H}$(?rhs$_9$)** which gives rise to one of rules in the preliminary implementation of **Append**. The derivation begins with the choice of the left hand side of the instance of the equation to be derived: This has to be an instance of **$\mathcal{H}$(APPEND(*c*, Insert(*d*,*i*)))**. Let us suppose σ is $\{d \mapsto$ **Create**$\}$.

Relevant Rewrite Rules of the Perturbed World

(10) **Append(q, Nullq) → q**
(14) **$\mathcal{H}$(Create) → Nullq**
(20) **$\mathcal{H}$(ENQUEUE(c, i))** → Enqueue($\mathcal{H}$(c), $\mathcal{H}$(i))})
(22) **$\mathcal{H}$(APPEND(c, d)) → Append($\mathcal{H}$(c), $\mathcal{H}$(d))**

Form of the theorem to be generated: **$\mathcal{H}$(APPEND(*c*, Insert(Create, *i*))) ≡ $\mathcal{H}$(?e)**
Normal form of **$\mathcal{H}$(APPEND(*c*, Insert(Create, *i*)))**: **Enqueue($\mathcal{H}$(*c*), $\mathcal{H}$(*i*))**
Rules used for the normal form:

Step (1) Invoke Synthesis Rule (1) on **$\mathcal{H}$(APPEND(*c*, Insert(Create, *i*)))**
$\mathcal{H}$(APPEND(*c*, Insert(Create, *i*))) ≡ Enqueue($\mathcal{H}$(*c*), $\mathcal{H}$(*i*))

Step (2) Expand Expression: **Enqueue($\mathcal{H}$(*c*), $\mathcal{H}$(*i*))**
Using Rule: (10)

$\mathcal{H}$(APPEND(*c*, Insert(Create, *i*))) ≡ Append(Enqueue($\mathcal{H}$(*c*), $\mathcal{H}$(*i*)), Nullq)

Step (3) Expand Expression: **Nullq**
Using Rule: (14)

$\mathcal{H}$(APPEND(*c*, Insert(Create, *i*))) ≡ Append(Enqueue($\mathcal{H}$(*c*), $\mathcal{H}$(*i*)), $\mathcal{H}$(Create))

Step (4) Expand Expression: **Enqueue(ℋ(*c*), ℋ(*i*))**
Using Rule: (20)

ℋ(APPEND(*c*, Insert(Create, *i*))) ≡ Append(ℋ(ENQUEUE(*c*, *i*)), ℋ(Create))

Step (5) Expand Expression: **Append(ℋ(ENQUEUE(*c*, *i*)), ℋ(Create))**
Using Rule: (22)

ℋ(APPEND(*c*, Insert(Create, *i*))) ≡ ℋ(APPEND(ENQUEUE(*c*, *i*), Create))

Step (6) Generalize the theorem in step (5) by replacing the constant
Create by the variable d to obtain the following equation:
ℋ(APPEND(*c*, Insert(*d*,*i*))) ≡ ℋ(APPEND(ENQUEUE(*c*, *i*), *d*))

Apply **Is-an-inductive theorem-of** on the above equation.
This yields **True** *confirming that the equation is a theorem.*

Hence the desired rule (obtained by dropping ℋ on both sides) is:
APPEND(*c*, Insert(*d*,*i*)) → APPEND(ENQUEUE(*c*, *i*), *d*)

4.4.3.2 The Predicate Is-an-inductive-theorem-of

Is-an-inductive-theorem-of is a procedure that is used for checking if an equation $e_1 \equiv e_2$ is a theorem of a convergent rewriting system S. The procedure is designed so that if it yields true on $e_1 \equiv e_2$, then $e_1 \equiv e_2$ is a theorem of S; if it yields false, then nothing can be said about $e_1 \equiv e_2$. While deriving a synthesis equation in the inductive theory, the procedure is used to check if a generalization of an equation is a theorem of PW. The procedure is described here.

The procedure is based on a method of using the KB-algorithm (see sec.3.3.3.1) for checking the convergence for proving inductive properties of a rewriting system. Suppose S is a convergent rewriting system. To check if $e_1 \equiv e_2$ is a theorem of S, perform the following steps:

(1) Form $S_1 = S \cup \{e_1 \rightarrow e_2 \text{ (or } e_2 \rightarrow e_1)\}$.

(2) Check if S_1 is convergent. The KB-algorithm of checking convergence (which consists of checking if every critical pair $\langle \alpha_1, \alpha_2 \rangle$ of S_1 is such that $\alpha_1\downarrow = \alpha_2\downarrow$) is used for this.

If the result of step (2) is affirmative, then $e_1 \equiv e_2$ is a theorem; otherwise nothing can be said about it, in general. Let us assume that there exists a procedure, called **Can-be-made-convergent**, that implements this method.

We will first briefly summarize the method, and then describe how **Is-an-inductive-theorem-of** is built on top of it.

The result that provides a basis for the above method is proved in **Theorem** 7 in Appendix III which gives a few useful results about convergent systems. The result is similar to the one that was first developed by Musser [38], and that has also been investigated in [22]. Our result is different because the cited works assume that S satisfies a notion of completeness (similar to the principle of definition) besides convergence.

In the present situation **PW**, whose theorems we are interested in, is convergent but does not satisfy the principle of definition. Because of this the above method is applicable only when e_1 (or e_2) is such that for every instantiation of the variables by generator constants, e_1 simplifies to a generator constant. The left hand side of every equation we wish to check is of the form $\mathfrak{H}(F(g_1, \ldots, g_n))$, where F is an implementing function symbol, and $g_1, \ldots, g_n$ are generator expressions. Note that $\mathfrak{H}(F(g_1, \ldots, g_n))$ reduces to $f(\mathfrak{H}(g_1, \ldots, g_n))$ by the $\mathfrak{H}$-rule corresponding to F. The latter expression satisfies the desired condition since f and $\mathfrak{H}$ are well-spanned[18] by **PW**.

There are several situations when the method described above is not applicable for proving an equation $e_1 \equiv e_2$. But there exists another equation $e_1' \equiv e_2'$ such that

---

18. Note that if a function f is well-spanned by **PW**, then every term of the form $f(t_1, \ldots, t_k)$, where $t_1, \ldots, t_k$ are generator terms, can be simplified to a generator term using **PW**.

(1) $e_1^! \equiv e_2^!$ can be proved using the above method,

(2) $e_1 \equiv e_2$ is a theorem if $e_1^! \equiv e_2^!$ is a theorem, and

(3) $e_1^! \equiv e_2^!$ can be derived automatically from $e_1 \equiv e_2$.

In other words, $e_1^! \equiv e_2^!$ is serving as a lemma for the theorem $e_1 \equiv e_2$. The procedure **Is-an-inductive-theorem-of** consists of transforming $e_1 \equiv e_2$ to $e_1^! \equiv e_2^!$, and then applying **Can-be-made-convergent** on $e_1^! \equiv e_2^!$. The transformation of $e_1 \equiv e_2$ to $e_1^! \equiv e_2^!$ is performed by a function $\mathcal{L}$, called the *lemma deriving function.* The lemma deriving function used by **Is-an_inductive-theorem-of** is defined below:

The Lemma Deriving Function ($\mathcal{L}$)

$\mathcal{L}$ is a function on expressions. $\mathcal{L}$ can be used to derive for a given equation $e_1 \equiv e_2$ a lemma that the proof of the former is dependent on. The two sides of the lemma are obtained by applying $\mathcal{L}$ to e_1 and e_2.

$\mathcal{L}$: **expression -> expression**

Usage: $\mathcal{L}(\alpha_1)$

Pre: α_1 is of the form $\mathcal{H}(\alpha_2)$, where α_2 does not contain the symbol $\mathcal{H}$.

Returns: An expression β that is obtained by replacing in $\alpha_1\downarrow$ every subexpression of the form $\mathcal{H}(d)$, where d is any terminal, by a new terminal d_1.

We will now illustrate the procedure **Is-an-inductive-theorem-of** to check if the equation $\mathcal{H}$(**APPEND**(c, **Insert**(d,i))) $\equiv$ $\mathcal{H}$(**APPEND**(**ENQUEUE**(c, i), d)) is a theorem of **PW** being used in our example. The equation was obtained in step (6) while deriving a synthesis equation in the previous section.

Equation to be checked: $\mathcal{H}$(**APPEND**(c, **Insert**(d,i))) $\equiv$ $\mathcal{H}$(**APPEND**(**ENQUEUE**(c, i), d)).

Step (1) Derive Lemma by applying $\mathcal{L}$:

(a) Simplify both sides,

(b) Replace $\mathcal{H}(c)$ by **q**, $\mathcal{H}(d)$ by **R**, $\mathcal{H}(i)$ by **i**

$\mathcal{H}$(APPEND(c, Insert(d, i)))

↓

Append($\mathcal{H}$(c), Add_at_head($\mathcal{H}$(d), $\mathcal{H}$(i)))

$\mathcal{H}$(APPEND(ENQUEUE(c, i), d))

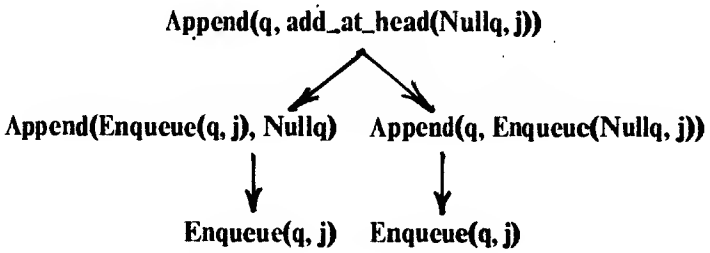
↓

Append(Enqueue($\mathcal{H}$(c), $\mathcal{H}$(i)), $\mathcal{H}$(d))

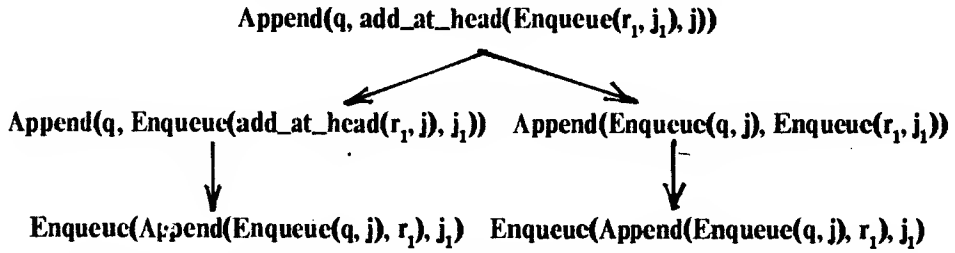
Lemma to be checked: **Append(q, Add_at_head(R, i)) ≡ Append(Enqueue(q, i), R)**

Step(2) Check if critical pairs are convergent:

(a) Critical pair determined by Rule (16):



(b) Critical pair determined by Rule (17):



4.4.3.3 An Instantiation for the Synthesis Equation

Here, we describe a method of finding a substitution σ that determines the left hand side of the instance of the theorem we wish to generate. Note that the left hand side of the theorem is already known to us which in the current example is $\mathcal{H}$(**APPEND(**c, **Insert(**d, i**)**)**)**. σ maps the terminals in the left hand side expression to suitable expressions. σ should be chosen so that the equation $\sigma(\mathcal{H}(\textbf{APPEND}(c, \textbf{Insert}(d, i)))) \equiv \sigma(\mathcal{H}(?e_2))$ is in the equational theory of **PW**. This implies that σ should be such that $\sigma(\mathcal{H}(\textbf{APPEND}(c, \textbf{Insert}(d, i))))$ and $\sigma(\mathcal{H}(?e_2))$ have the same normal form. Note that $\mathcal{H}(?e_2)$ is unavailable to us at the moment. So, σ has to be determined from the left hand side expression alone. Since the theorem $\mathcal{H}(\textbf{APPEND}(c, \textbf{Insert}(d, i))) \equiv \mathcal{H}(?e_2)$ is not necessarily in the equational theory of **PW**, an arbitrary substitution that maps terminals to generator terms cannot be used.

The following fact about our proof method (for inductive properties) serves as the

basis for the method of finding σ. The basis step of the inductive proof can always be carried out using the equational logic. So, we choose the σ that corresponds to a basis step of the proof of the lemma. The instantiation corresponding to the basis step can be determined automatically starting from the left hand side of the theorem alone.

Finding such a σ involves two stages because the proof of the theorem, as you may recall, involves two stages: Converting the theorem to the lemma, and then proving the lemma itself. We first determine a substitution ω that corresponds to a basis step of the proof of the lemma. σ is determined from ω using the method used by the lemma defining function $\mathcal{L}$ to convert the theorem to the lemma. We describe the two steps below.

Step (1) Determination of ω

(a) Find the left hand side of the lemma.

This is obtained by applying $\mathcal{L}$, the lemma defining function, to the left hand side of the theorem. For our example: Left hand side of the theorem is $\mathcal{H}$(**APPEND(*c*, Insert(*d*, *i*))**). To obtain the left hand side of the lemma, we simplify the expression, and replace every subexpression that has $\mathcal{H}$ at the root by a new terminal: $\mathcal{H}$(**APPEND(*c*, Insert(*d*, *i*))**) $\rightarrow^*$ **Append($\mathcal{H}$(c), Add_at_head($\mathcal{H}$(*d*), $\mathcal{H}$(*i*)))**. So the left hand side of the lemma is **Append(q, Add_at_head(R, i))**.

(b) Find a basis step in the proof of the lemma

For this, compute all the superpositions between the left hand sides of the rules of **PW** and the left hand side of the lemma. Simplify the superpositions. A sufficient condition for a superposition to correspond to a basis step is that its normal form is a generator expression. The most general unifier that determines such a superposition is a candidate σ. The following table gives the result of performing the above steps on the current example. The columns, in order, give the rewrite rule in **PW** responsible for the superposition, the superposition, and the normal form of the superposition. The first superposition in the list simplifies to a generator expression. Therefore, ω is the most general unifier corresponding to the first superposition, which is **{R $\mapsto$ Nullq}**.

Rule	*Superposition*	*(Superposition)↓*
(16)	Append(*q*, Add_at_head(Nullq, *i*))	Enqueue(*q*, *i*)
(17)	Append(*q*, Add_at_head(	Enqueue(Append(*q*,
	Enqueue(r_1, j_1), *i*))	Add_at_head(r_1, *i*)), j_1)

Step (2) Determine σ from ω

ω provides instantiations for the terminals in the left hand side of the lemma. σ instantiates the terminals in the left hand side of the theorem. Our objective is to find a σ so that when the left hand sides (of the lemma and the theorem) are instantiated by σ and ω, respectively, they simplify to the same expression.

For instance, in the current example, the left hand side of the theorem is e_1 = $\mathcal{H}$(APPEND(*c*, Insert(*d*, *i*))), whose normal form is e_2 = Append($\mathcal{H}$(c), Add_at_head($\mathcal{H}$(*d*), $\mathcal{H}$(*i*))). The left hand side of the lemma is e_3 = Append(q, Add_at_head(R, i)), which was obtained by replacing $\mathcal{H}$(*d*) by *r*, and $\mathcal{H}$(*c*) by *q*. ω maps *r* to Nullq, and leaves the rest of the terminals unchanged. Therefore, σ should map *d* to an expression such that Nullq $\equiv$ $\mathcal{H}$(*d*) is a theorem in the equational theory of PW. Therefore, the instantiation for *d* can be determined using the first two synthesis rules by generating a theorem that has Nullq on the left hand side, and an expression of the form $\mathcal{H}$(?e) on the right hand side. The generation sequence is shown below. The first theorem is obtained by invoking Synthesis Rule (1) for the expression Nullq. The second theorem is obtained by using Synthesis Rule (2); rewrite rule (14) of PW is used for expand. The right hand side, $\mathcal{H}$(Create), of the theorem generated determines σ as $\{d \mapsto$ Create$\}$.

Nullq $\equiv$ Nullq

$\equiv$ $\mathcal{H}$(Create)

4.5 An Abstract Implementation of the Derivation Procedure

Below, we give an implementation for a procedure **Generate-a-rule**. The procedure determines a suitable right hand side expression for a rewrite rule in a partial preliminary implementation given the left hand side expression. The procedure also expects a Perturbed World and a termination ordering as inputs. The procedure is implemented in a high level

algorithmic language whose semantics is self-explanatory.

The implementation assumes that there exist two procedures **Is-an-inductive-theorem-of** and **A-suitable-instantiation-for-lhs**. The latter finds a suitable substitution that determines the instance of synthesis equation to be generated.

The procedure performs essentially the theorem generation illustrated before in a systematic fashion. Roughly, it operates as follows. It finds the instance of the left hand side of the synthesis equation by applying **A-suitable-instantiation-for-lhs** to 𝔊(lhs). It simplifies this expression to its normal form. The normal form is then expanded repeatedly using appropriate rewrite rules of **PW** until a suitable right hand side is encountered.

The nontrivial aspect of the procedure concerns performing expansion in an effective fashion. There are two problem areas. Firstly, expansion is not uniformly terminating. That is, expansion is a potentially nonterminating activity. The procedure uses the termination ordering $\succ$ to circumvent this problem. The right hand side has to be an expression that is less than the given left hand side. But, expanding an expression always gives rise to a bigger expression in the ordering $\succ$. Thus, the procedure can be terminated the moment we encounter an expression that is not less than the left hand side. (Note that the $\succ$ is such that there can only be a finite number of expressions less than any given expression.)

Secondly, expansion is not uniquely terminating. That is, an expression can be expanded in several different (but finitely many, because there are only finite number of rules in **PW**) ways using the rules in **PW**. All of them do not necessarily lead to the same final expression. Some of them may not even lead to a suitable right hand side expression. In the examples illustrated earlier, the rules of **PW** were carefully chosen so that they resulted in the desired right hand side. A working implementation, however, is forced to keep track of all possible expansions since any one of them can result in the desired right hand side. In the implementation given below the variable S is used for this purpose.

This chore, in fact, happens to be the main source of inefficiency in the synthesis procedure. We use the following obvious ways of getting rid of unproductive expansion paths. Firstly, type information is used to eliminate some of the candidate rewrite rules for expansion. Secondly, expansions that result in an expression that is not less than the left hand

side are not going to be fruitful. Finally, we make a distinction between the variables that appear in the rewrite rules of PW, and the ones in the given left hand side. The latter, which are terminals, are treated as constants. This eliminates several rewrite rules for expansion that are candidates otherwise.

It should be noted that the procedure given below is only a part of a complete implementation of the synthesis procedure. The other part is expected to determine the left hand side of the rules. We have assumed that there exists a procedure to determine the left hand sides. If the following procedure does not succeed in finding a suitable right hand side for a given left hand side, then another set of left hand sides have to be generated, and the following procedure reexecuted.

```
Generate-a-rule = proc ( PW: Perturbed World, lhs: F(g1, . . . , gn),
                                                ≻: ordering) returns (Rewrite Rule)

%Initialization
σ: Substitution ← A-suitable-instantiation-for-lhs
ilhs ← σ(lhs)
S ← {ℑ(ilhs)↓}

repeat

%Test if expansion can be stopped
if There-exists-a-suitable-candidate-in(S)
then rhs ← Fetch-a-suitable-candidate-from(S)
     return(lhs → rhs)
     endif

%If a candidate has not been generated yet, expand by one more step
S1 ← Φ
for every t ∈ S do
        S1 ← S1 ∪ set-of-all-expansions-of t by PW
        endfor
S ← S1

%Drop from S1 unproductive expressions
for every t ∈ S1 do
        if ~(lhs ≻ t) then S1 ← S1 - {t}

forever
```

%Subprocedure description

There-exists-a-suitable-candidate-in: subproc (S: Set[Expression]) returns (Boolean)

if $\exists$ t $\in$ S **such that**
  $\exists$ $\mathcal{H}(F(g_1, \ldots, g_n) \equiv \mathcal{H}(?rhs) \in$ **Gen[ilhs** $\equiv$ **t]** such that
  (1) ?rhs does not contain $\mathcal{H}$ or operations of the implemented type,
  (2) $F(g_1, \ldots, g_n) \succ$?rhs, and
  (3) **Is-an-inductive-theorem-of-PW**($\mathcal{H}(F(g_1, \ldots, g_n) \equiv \mathcal{H}(?rhs)$)
then return(True) else return(False)

end subproc

%Subprocedure description

Fetch-a-suitable-candidate-from: subproc (S: Set[Expression]) returns (Expression)

if $\exists$ t $\in$ S **such that**
  $\exists$ $\mathcal{H}(F(g_1, \ldots, g_n) \equiv \mathcal{H}(?rhs) \in$ **Gen[ilhs** $\equiv$ **t]** such that
  (1) ?rhs does not contain $\mathcal{H}$ or operations of the implemented type,
  (2) $F(g_1, \ldots, g_n) \succ$?rhs, and
  (3) **Is-an-inductive-theorem-of-PW**($\mathcal{H}(F(g_1, \ldots, g_n) \equiv \mathcal{H}(?rhs)$)
then return(t)
end subproc

---

end Generate-a-rule

---

set-of-all-expansions-of_by: Expression X Rule -> Set[Expression]
Usage: **set-of-all-expansions-of t by** $\gamma \rightarrow \delta$
Returns: Returns the set of all possible expansions of a given term via a given rule.

---

set-of-all-expansions-of_by: Expression X Set[Rule] -> Set[Expression]
Usage: **set-of-all-expansions-of t by** $\mathfrak{R}$
Returns: The set of all terms s such that
  s = $\cup$ **set-of-all-expansions-of t by** *R*, for all $R \in \mathfrak{R}$

---

expand_in_by: Expression X Occurrence X Rule -> Expression
Usage: **expand** t_1 **in** *u* **by** $\gamma \rightarrow \delta$

Pre: $Varset(t_1) \cap Varset(\gamma) = \Phi$ %For convenience

t_1/u *is-unifiable-with* δ

Returns: **expand** t_1 **in** u **by** $\gamma \rightarrow \delta$ yields a term t_2 such that every term that reduces (in u by $\gamma \rightarrow \delta$) to an instance of t_1 will be an instance of t_2. In other words t_2 is the most general instance of all the terms that reduce (in u by $\gamma \rightarrow \delta$) to an instance of t_1. Note that the result the function returns is unique upto permutations of the variables. This is because σ, which is the most general unifier of two terms, is always unique when restricted to the variables in the two terms t_1 and δ.

---

expands-to_in_by: Expression X Expression X Occurrence X Rule -> Bool

Usage: t_1 **expands-to** t_2 **in** u **by** $\gamma \rightarrow \delta$

Pre: $Varset(\gamma) \cap Varset(t_1) = \Phi$

Returns: A predicate that tests if a term expands to another given term.

(t_1/u) *is-unifiable-with* $\delta \wedge t_2 =$ **expand** t_1 **in** u **by** $\gamma \rightarrow \delta$

---

5. Extending the Derivation Problem

The derivation problem and the derivation procedure described in the last chapter apply to a situation in which the representing domain ($\mathfrak{R}$) for the desired preliminary implementation is unrestricted. That is, $\mathfrak{R}$ includes all the values of the representation type. This section extends the problem to the more general situation where $\mathfrak{R}$ is a subset of the value set of the representation type.

$\mathfrak{R}$ contains the set of values that are permitted to be used by a preliminary implementation for representing the values of the implemented type. It is characterized by the association specification supplied by the user. Suppose $\mathcal{A}$ and $\mathfrak{I}$ are the abstraction function and the invariant specified by the association specification respectively. Then $\mathfrak{R}$ is the set of all values for which $\mathfrak{I}$ is true. The present situation is one in which $\mathfrak{I}$ is true on only a subset of the representation value set.

For instance, consider the association specification given in Fig. 15. This example will be used to illustrate the procedure described in the chapter. It specifies an implementation of **Queue_Int** interms of **Array_Int X Integer X Integer**. The abstraction function $\mathcal{A}$ can be described informally as follows. **Nullq** can be represented by any triple in which both the integer components are equal. A nonempty queue can be represented by a triple **<v, i, j>**. **v** is an array of arbitrary length containing the elements of the queue, in order, between the index values **i** and **j**-1. In other words, **i** points to the front end of the queue, and **j** points to the next availabe position in **v** for adding a new element into the queue. The invariant $\mathfrak{I}$ is true on all triples such that $\mathbf{i} \leq \mathbf{j}$ and the array is guaranteed to be defined on all

---

Fig. 15. Queue_Int in terms of Triple

$\mathcal{A}$(<v, i, i>) ≡ Nullq
$\mathcal{A}$(<Assign(v, e, j), i, j+1>) ≡ if i = j+1 then Nullq
else Enqueue($\mathcal{A}$(<v, i, j>), e)

$\mathfrak{I}$(<v, i, i>) ≡ True
$\mathfrak{I}$(<Assign(v, e, j), i, j+1>) ≡ if i = j+1 then True
else if i ≤ j+1 then $\mathfrak{I}$(<v, i, j>)
else False

index values between **i** and **j**.

5.1 Characterization of the Problem

The criterion of correctness (stated in the previous chapter in Sec 4.2.1) that was used to characterize the problem earlier is applicable in the current situation as well. For convenience, we repeat the criterion below: A preliminary implementation of a data type is correct with respect to an association specification (that characterizes an abstraction function $\mathcal{A}$, and a representing domain $\mathfrak{R}$) if the following properties hold.

(1) *Totality Property:* Every implementing function is total over $\mathfrak{R}$.

(2) *Homomorphism Property:* The implementing function **F** and the operation **f** of the implemented type are related by the following homomorphism property:
$(\forall\ r \in \mathfrak{R})[\mathcal{H}(F(..., r, ...)) = f(..., \mathcal{H}(r), ...)]$, where $\mathcal{H}$ is a function defined as:
$\mathcal{H}(r) = \mathcal{A}(r)$ if $r \in \mathfrak{R}$
r otherwise

Based on the above criterion, the derivation of a preliminary implementation was viewed earlier as a problem of finding a set of rewrite rules **PI** so that **PI** ∪ **IW** and **PI** ∪ **PW** satisfy the principle of definition. We still view the problem the same way. But, now the implementing functions need be defined only on the values in $\mathfrak{R}$, and the homomorphism property need only be verified on the values in $\mathfrak{R}$. This means that **PI** ∪ **IW** and **PI** ∪ **PW** need satisfy the principle of definition only with respect to a subset[19] of the set of all generator constants of the representation type. This subset is the representing domain of constants T characterized by the association specification as follows: $T = \{ t \mid \mathcal{I}(t) \equiv \textbf{True} \}$. A proof of the claim that if **PI** ∪ **IW** and **PI** ∪ **PW** satisfy the principle of definition with respect to T, then **PI** is correct can be carried out along the same lines as the proof of the **Correctness Theorem** (Sec. 4.2.2). The proof for the present case can be obtained by

---

19. A system **S** satisfies the principle of definition with respect to **T** if the every constant of the form $F(g_1, \ldots, g_n)$, where **F** is a nongenerator function symbol and $g_1, \ldots, g_n$ are generatore constants in **T**, has a unique normal (in **S**) that is a generator constant in **T**.

systematically replacing in the earlier proof the phrase "the principle of definition" by the phrase "the principle of definition with respect to T".

5.2 Derivation of a Preliminary Implementation

First we formulate the synthesis conditions that are used as a guide in the derivation of a preliminary implementation, and then describe a procedure to derive a set of rewrite rules **PI** that satisfies the synthesis conditions. The synthesis conditions are sufficient to ensure that **PI** ∪ **IW** and **PI** ∪ **PW** satisfy the principle of definition with respect to **T**.

5.2.1 The Synthesis Conditions

The synthesis conditions for a preliminary implementation **PI** are the following:

(1) **Totality Condition:**

(a) **PI** is well-spanned with respect to **T** (for every implementing function) with every rule in it being of the form $F(g_1, \ldots, g_n) \rightarrow t$, where **F** is an implementing function symbol, and $g_1, \ldots, g_n$ are generator expressions.

(b) **PI** has the uniform termination property.

(2) **Uniqueness Condition: PI** has the unique termination property.

(3) **Homomorphism Condition:** For every rule $F(g_1, \ldots, g_n) \rightarrow t$ in **PI**, $\mathfrak{I}(g_1) \wedge \ldots \wedge \mathfrak{I}(g_n)$[20] $\Rightarrow \mathfrak{H}(F(g_1, \ldots, g_n)) \equiv \mathfrak{H}(t)$ is a theorem of **PW**.

(4) **Invariance Condition:** For every rule $F(g_1, \ldots, g_n) \rightarrow t$ in **PI**, where the range of **F** is the representation type, $\mathfrak{I}(g_1) \wedge \ldots \wedge \mathfrak{I}(g_n) \Rightarrow \mathfrak{I}(t) \equiv$ **True** is a theorem.

It is interesting to note the effect of the presence of the invariant $\mathfrak{I}$ on the synthesis

---

20. Here, we assume that each of the expressions $g_1, \ldots, g_n$ is of the representation type. If not, the antecedent would consist of a conjunction of $\mathfrak{I}$ applied to only those expressions among $g_1, \ldots, g_n$ that are of the representation type. The same qualification applies to condition (4), as well.

conditions. The Totality Condition and the Uniqueness Condition remain as before, and serve the same purpose: The Totality Condition ensures that an implementing function is defined and terminates on every value in the representing domain. The Uniqueness Condition ensures that an implementing function yields a unique value on every argument. The Homomorphism Condition, which ensures that every implementing function satisfies the homomorphism property, now requires that $\mathcal{H}(F(g_1, \ldots, g_n)) \equiv \mathcal{H}(t)$ be a theorem only under the assumption that the arguments to F satisfy $\mathcal{I}$. The Invariance Condition imposes an additional constraint on the expression that may appear on the right hand side of a rule: It ensures that every implementing function preserves $\mathcal{I}$. The **Synthesis Theorem** to follow shows that when **PI** satisfies all the synthesis conditions **PI** ∪ **IW** and **PI** ∪ **PW** satisfy the principle of definition with respect to **T**.

The Synthesis Theorem

Theorem 2 Let **PI** be a set of rewrite rules that satisfies all the synthesis conditions. Then, **PI** ∪ **IW** and **PI** ∪ **PW** satisfy the principle of definition with respect to **T**, where **T** is the representing domain of constants characterized by the invariant $\mathcal{I}$.

Proof Appendix III

5.2.2 Deriving the rules of PI

The derivation **PI** follows the same general pattern as before. The first task is to construct the **PW** which is done as before by combining the specification of the implemented type, the homomorphism specification, and any desired parts of the specifications of the implementing types. The homomorphism specification is derived from the abstraction function specification as before (sec. 4.2.2). For instance, **PW** for the example under consideration is given in Fig. 16. Note that **PW** does not contain the invariant specification. The infomation pertaining to the invariant will be maintained as a different entity. This will be explained shortly.

The rules of **PI** are derived so that every synthesis condition except the Uniqueness

Fig. 16. The Perturbed World.

(1) Front(Nullq) → ERROR
(2) Front(Enqueue(Nullq, e)) → e
(3) Front(Enqueue(Enqueue(q, e1), e2)) → Front(Enqueue(q, e1))

(4) Dequeue(Nullq) → ERROR
(5) Dequeue(Enqueue(Nullq, e)) → Nullq
(6) Dequeue(Enqueue(Enqueue(q, e1), e2)) → Enqueue(Dequeue(Enqueue(q, e1)), e2)

(10) Append(q, Nullq) → q
(11) Append(q1, Enqueue(q2, e2)) → Enqueue(Append(q1, q2), e2)

(12) Empty(Nullq) → True
(13) Empty(Enqueue(q, e)) → False

(14) $\mathcal{H}$(<v, i, i>) → Nullq
(15) $\mathcal{H}$(<Assign(v, e, j), i, j+1>) → if i = j+1 then Nullq
else Enqueue($\mathcal{H}$(<v, i, j>), $\mathcal{H}$(e))

(16) $\mathcal{H}$(NULLQ()) → Nullq
(17) $\mathcal{H}$(ENQUEUE(c, i)) → Enqueue($\mathcal{H}$(c), $\mathcal{H}$(i))
(18) $\mathcal{H}$(DEQUEUE(c)) → Dequeue($\mathcal{H}$(c))
(19) $\mathcal{H}$(APPEND(c1, c2)) → Append($\mathcal{H}$(c1), $\mathcal{H}$(c2))
(20) $\mathcal{H}$(EMPTY(c)) → Empty($\mathcal{H}$(c))

(21) $\mathcal{H}$(if_then_else(b, v_1, v_2)) → if_then_else(b, $\mathcal{H}$(v_1), $\mathcal{H}$(v_2))

Condition is met. The procedure derives the preliminary implementation for one operation at a time by deriving a separate set of rewrite rules for every operation. The method used is the same for every operation. The procedure first determines the left hand sides of all the rules to derive a partial preliminary implementation. Then, it determines a suitable right hand side for each of the rules in the partial preliminary implementation.

5.2.2.1 Determining the Left Hand Side

The technique used for determining the left hand sides is the same as before because the Totality Condition, which is used for the purpose, is the same as before. The left hand sides are derived so that the set of expressions appearing as arguments to every implementing function is well-spanned.[21] Fig. 17 gives a possible set of left hand sides for a preliminary implementation for the example under consideration. As before, we use the question mark identifiers as place holders for expressions to be determined yet.

Fig. 17. A Partial Preliminary Implementation

Representation

Array_Int X Integer X Integer

Definitions

NULLQ() → $?rhs_1$

ENQUEUE(<v, i, j>, e) → $?rhs_2$

FRONT(<v, i, i>) → $?rhs_3$
FRONT(<Assign(v, e, j), i, j+1>) → $?rhs_4$

DEQUEUE(<v, i, i>) → $?rhs_4$
DEQUEUE(<Assign(v, e, j), i, j+1>) → $?rhs_5$

APPEND(<v1, i1, j1>, <v2, i2, i2>) → $?rhs_6$
APPEND(<v1, i1, j1>, <Assign(v2, e, j2), i2, j2 + 1>) → $?rhs_7$

EMPTY(v, i, j>) → $?rhs_8$

---

21. Note that if a set is well-spanned, then it is well-spanned with respect to any set of generator constants.

5.2.2.2 Determining the Right Hand Side

The general strategy used to derive the right hand sides is the same as before. They are derived so that the Homomorphism Condition, the Invariance Condition, and the second part of the Totality Condition (which is left unensured while determining the left hand side) are ensured. The right hand side of a rule is determined by deriving a *synthesis equation* corresponding to the rule. A synthesis equation corresponding to a rule $F(g_1, \ldots, g_n) \rightarrow ?t$ is an equation of the form $\mathcal{H}(F(g_1, \ldots, g_n)) \equiv \mathcal{H}(qt)$ that satisfies the following conditions:

(1) $\mathcal{I}(g_1) \wedge \ldots \wedge \mathcal{I}(g_n) \Rightarrow \mathcal{H}(F(g_1, \ldots, g_n)) \equiv \mathcal{H}(?t)$ is a theorem of **PW**.

(2) If the range type of F is the representation type, then
$\mathcal{I}(g_1) \wedge \ldots \wedge \mathcal{I}(g_n) \Rightarrow \mathcal{I}(t) \equiv$ **True** is a theorem of **PW**.

(3) $F(g_1, \ldots, g_n) \succ ?t$, $\succ$ is the termination ordering on expressions.

(4) $?t$ may only contain only the permitted operation symbols of the implementing types, and the implementing function symbols.

Note that the synthesis equations have additional constraints here because of $\mathcal{I}$. So, the derivation of the synthesis equations is going is going to have to be performed slightly differently. This is the topic of the next section.

5.3 Deriving the Synthesis Equations

The general strategy used for deriving a synthesis equation is the same as before. That is, we generate a series of theorems of **PW** until we encounter one that qualifies to be a synthesis equation. We use the same pair of synthesis rules for generating the theorems of **PW**. The only difference lies in the set of rewrite rules used for expansion while generating the theorems. Earlier, the rewrite rules in **PW** were used. But now, it is necessary to use an additionalset of rewrite rules.

There are two reasons for this. Firstly, a synthesis equation $\mathcal{H}(F(g_1, \ldots, g_n)) \equiv \mathcal{H}(?t)$ to be derived is a theorem of **PW** in a special context: A context determined by the fact that $g_1, \ldots, g_n$ satisfy the invariant $\mathcal{I}$. In deriving the synthesis

equations, one has to use rewrite rules describing this context besides the rewrite rules in **PW**. Secondly, **?t** has to be determined so that $\mathfrak{I}$(**?t**) ≡ **True** is a theorem. For this, it is necessary to use the rewrite rules in the specification of $\mathfrak{I}$. These additional rewrite rules, which describe information pertaining to the invariant, are maintained as a separate entity called the *Temporary World* (**TW**). We will discuss more about **TW** - its composition, and its construction - later. It is sufficient to say the following at this point: **TW** consists of rules that specify $\mathfrak{I}$, and rules that assert that $\mathbf{g}_1, \ldots, \mathbf{g}_n$ satisfy the invariant. The rules in **TW** are used for expansion as well as to ensure that **?t** satisfies $\mathfrak{I}$.

It should be noted that part of the Temporary World used in the derivation of a preliminary implementation could be different for different rules in the preliminary implementation. This is because the argument expressions appearing on the left hand side $(\mathbf{g}_1, \ldots, \mathbf{g}_n)$ are usually different for different rules. Consequently, the part of **TW** that changes has to be constructed afresh at the beginning of the derivation of every rule. (The temporary life time of a part of **TW** is what prompted us to name **TW** a Temporary World.)

5.3.1 A Simple Illustration

In the following, we show the derivation of a synthesis equation corresponding to the rewrite rule **ENQUEUE**($\langle v, i, j\rangle$, *e*) → $\mathbf{?rhs}_2$ in the partial preliminary implementation shown in Fig. 17; The derivation provides an illustration of how the generation of theorems is influenced by **TW**. It also illustrates for the first time performing expansion using rewrite rules that have conditional expressions in them.

The **TW** used for the derivation is shown below. For ease of reference, also given below are rules excerpted from **PW** (Fig. 16) that are relevant in the present derivation. Rules numbered (9) and (10) in **TW** are the specification of $\mathfrak{I}$. The rule numbered (11) asserts that the argument $\langle v, i, j\rangle$ to **ENQUEUE** satisfies $\mathfrak{I}$. The fourth rule is a property of the invariant: Any triple $\langle v, i, j\rangle$ that satisfies $\mathfrak{I}$ is such that $i \leq j$. This can be proved as a theorem from the specification of $\mathfrak{I}$. We will see how this is obtained in a subsequent section where we discuss more about the Temporary World.

The Relevant Rules of PW

(1) $\mathcal{H}(\langle v, i, i\rangle) \rightarrow$ Nullq

(2) $\mathcal{H}(\langle$Assign(v, e, j), i, j+1$\rangle) \rightarrow$ if i = j+1 then Nullq
else Enqueue($\mathcal{H}(\langle v, i, j\rangle)$, $\mathcal{H}$(e))

(3) $\mathcal{H}$(ENQUEUE(x, e)) $\rightarrow$ Enqueue($\mathcal{H}$(x), $\mathcal{H}$(e))

(4) if_then_else(False, v1, v2) $\rightarrow$ v2

(5) if_then_else(True, v1, v2) $\rightarrow$ v1

(6) $\mathcal{H}$(if_then_else(b, v1, v2)) $\rightarrow$ if_then_else(b, $\mathcal{H}$(v1), $\mathcal{H}$(v2))

(7) $x = y+1 \rightarrow$ not($x \leq y$)

(8) not(True) $\rightarrow$ False

The Temporary World

(9) $\mathcal{I}(\langle v, i, i\rangle) \rightarrow$ True

(10) $\mathcal{I}(\langle$Assign(v, e, j), i, j+1$\rangle) \rightarrow i \leq j+1 \wedge [i = j+1 \vee \mathcal{I}(\langle v, i, j\rangle)]$

(11) $\mathcal{I}(\langle v, i, j\rangle) \rightarrow$ True

(12) $i \leq j \rightarrow$ True

Shown below is a generation of a series of theorems by invoking the synthesis rules using the rewrite rules shown above for expansion. The generation results in the derivation of a synthesis equation of the form we desire. The first theorem in the series is obtained by invoking Synthesis Rule (1) for the expression $\mathcal{H}$(ENQUEUE($\langle v, i, j\rangle$, e)); the normal form of this expression is Enqueue($\mathcal{H}(\langle v, i, j\rangle)$, $\mathcal{H}(e)$). The rest of the theorems in the series are obtained by invoking Synthesis Rule (2) using different rules in PW and TW for expansion.

An explanation about our choice of the rewrite rules for expansion in the following derivation is in order. Recall that the ultimate objective of expansion is to drive the symbol $\mathcal{H}$ in the right hand side of the equation in Step (1) to the outermost level of the expression. Inspection of the rules of PW reveals two possible sets of rules which could be used for this purpose. The first one is the $\mathcal{H}$-rules, in particular, Rule (3) of PW; however, applying this rule in Step (1) will yield an expression identical to the one on the left hand side which is not acceptable. The other possibility is applying the rules of the homomorphism specification, i.e., either Rule (1) or (2) of PW. Rule (1) is clearly not applicable. Rule (2) is also not applicable. A closer look, however, reveals that Enqueue($\mathcal{H}(\langle v, i, j\rangle)$, $\mathcal{H}(e)$) has the form of the expression in the else-arm of the conditional expression on the right hand side of

Rule (2). Hence, we make an attempt to expand **Enqueue**($\mathcal{H}$(<*v*, *i*, *j*>), $\mathcal{H}$(*e*)) to an expression of the form **if_then_else**(..., ..., **Enqueue**($\mathcal{H}$(<*v*, *i*, *j*>), $\mathcal{H}$(*e*))). The manipulations performed in Steps (2) through (4) are precisely aimed at this.

Form of synthesis equation to be derived: $\mathcal{H}$(ENQUEUE(<*v*, *i*, *j*>, *e*))
Normal form of $\mathcal{H}$(ENQUEUE(<*v*, *i*, *j*>, *e*)): **Enqueue**($\mathcal{H}$(<*v*, *i*, *j*>), $\mathcal{H}$(*e*))
Rules used for simplification:

Step (1) Invoke Synthesis Rule (1) on $\mathcal{H}$(ENQUEUE(<*v*, *i*, *j*>, *e*))
$\mathcal{H}$(ENQUEUE(<*v*, *i*, *j*>, *e*)) ≡ **Enqueue**($\mathcal{H}$(<*v*, *i*, *j*>), $\mathcal{H}$(*e*))

Step (2) Expand Expression: **Enqueue**($\mathcal{H}$(<*v*, *i*, *j*>), $\mathcal{H}$(*e*))
Using Rule: (4)

$\mathcal{H}$(ENQUEUE(<*v*, *i*, *j*>, *e*)) ≡ **if False then v1 else Enqueue**($\mathcal{H}$(<*v*, *i*, *j*>), $\mathcal{H}$(*e*))

Step (3) Expand Expression: **False**
Using Rule: (8)

$\mathcal{H}$(ENQUEUE(<*v*, *i*, *j*>, *e*)) ≡ **if ~(True) then v1 else Enqueue**($\mathcal{H}$(<*v*, *i*, *j*>), $\mathcal{H}$(*e*))

Step (4) Expand Expression: **True**
Using Rule: (12)

$\mathcal{H}$(ENQUEUE(<*v*, *i*, *j*>, *e*)) ≡ **if not**($i \leq j$) **then v1 else Enqueue**($\mathcal{H}$(<*v*, *i*, *j*>), $\mathcal{H}$(*e*))

Step (5) Expand Expression: ~($i \leq j$)
Using Rule: (7)

$\mathcal{H}$(ENQUEUE(<*v*, *i*, *j*>, *e*)) ≡ **if** $i = j+1$ **then v1 else Enqueue**($\mathcal{H}$(<*v*, *i*, *j*>), $\mathcal{H}$(*e*))

Step (6) Expand Expression: **if** $i = j+1$ **then v1 else Enqueue**($\mathcal{H}$(<*v*, *i*, *j*>), $\mathcal{H}$(*e*))
Using Rule: (2)

$\mathcal{H}$(ENQUEUE(<*v*, *i*, *j*>, *e*)) ≡ $\mathcal{H}$(<**Assign**(*v*, *e*, *j*), *i*, $j+1$>)

Note that the right hand side of the last theorem in the above series is such that

$$\text{ENQUEUE}(\langle v, i, j\rangle, e) \succ \langle \text{Assign}(v, e, j), i, j+1\rangle$$
$$\mathfrak{I}(\langle \text{Assign}(v, e, j), i, j+1\rangle) \rightarrow^* \text{True}$$

Hence, we have the following preliminary implementation for ENQUEUE:
$\text{ENQUEUE}(\langle v, i, j\rangle, e) \rightarrow \langle \text{Assign}(v, e, j), i, j+1\rangle$

Let us, for a moment, draw the attention of the reader back to steps (2) through (4) in the above derivation. Their aim was merely to expand $\textbf{Enqueue}(\mathfrak{H}(\langle v, i, j\rangle), \mathfrak{H}(e))$ to a conditional expression that had the former expression as its **else**-arm. The purpose of such a transformation was to make it possible to apply (for expanding) a rewrite rule that had a conditional expression on the right hand side. A situation such as this is encountered commonly during the generation of theorems. This is especially so when the rules of the input specifications have conditional expressions in them. Hence it is useful to extend the definition of the mechanism **expand** so that rewrite rules with conditional expressions on their right hand side can be applied directly to an expression that is not a conditional expression. We describe the extension below. In future illustrations of the derivation of synthesis equations, we will be using the extended version of **expand**.

Suppose $\mathbf{e_1} \rightarrow \textbf{if\_then\_else}(\mathbf{b}, \mathbf{e_{21}}, \mathbf{e_{22}})$ is a rewrite rule, and α is an expression that is being expanded by using the former rule. According to the existing definition of **expand**, the following protocol is used for expanding α:

Protocol 1:

(1) Check if α (or a subexpression in it) is unifiable with $\textbf{if\_then\_else}(\mathbf{b}, \mathbf{e_{21}}, \mathbf{e_{22}})$; if so, let θ be the most general unifier.

(2) Replace $\theta(\alpha)$ (or the subexpression in it) by $\theta(\mathbf{e_1})$

Note that according to the above protocol α is expandible only if α (or a subexpression in it) is of the form **if_then_else**(...). Now, we introduce two additional ways in which the rule can be used for expansion.

Protocol 2:

(1) Check if α (or a subexpression in it) is unifiable with $\mathbf{e}_{21}$; if so, let θ be the most general unifier.

(2) Check if $\theta(\mathbf{b}) \rightarrow^*$ **True**, or $\sim(\theta(\mathbf{b})) \rightarrow^*$ **False**.

(3) If so, replace $\theta(\alpha)$ (or a subexpression in it) by $\theta(\mathbf{e}_1)$.

Protocol 3:

(1) Check if α (or a subexpression in it) is unifiable with $\mathbf{e}_{22}$; if so, let θ be the most general unifier.

(2) Check if $\theta(\mathbf{b}) \rightarrow^*$ **False**, or $\sim(\theta(\mathbf{b})) \rightarrow^*$ **True**.

(3) If so, replace $\theta(\alpha)$ (or a subexpression in it) by $\theta(\mathbf{e}_1)$.

Using Protocol 3, the preliminary implementation of **Enqueue** derived earlier can be obtained in just two steps as shown below. The theorem in step (1) is obtained as before. The theorem in the second step is obtained by using Rule (2) of **PW** for expansion under protocol (3). Note that the boolean expression under consideration is $i = j+1$; $i = j+1 \rightarrow^*$ **False** by Rules (7), (12) and (8).

Form of synthesis equation to be derived: $\mathfrak{H}$(ENQUEUE(⟨*v*, *i*, *j*⟩, *e*))
Normal form of $\mathfrak{H}$(ENQUEUE(⟨*v*, *i*, *j*⟩, *e*)): Enqueue($\mathfrak{H}$(⟨*v*, *i*, *j*⟩), $\mathfrak{H}$(*e*))
Rules used for simplification:

Step (1) Invoke Synthesis Rule (1) on $\mathfrak{H}$(ENQUEUE(⟨*v*, *i*, *j*⟩, *e*))
$\mathfrak{H}$(ENQUEUE(⟨*v*, *i*, *j*⟩, *e*)) ≡ Enqueue($\mathfrak{H}$(⟨*v*, *i*, *j*⟩), $\mathfrak{H}$(*e*))

Step (2) Expand: Occurrence: λ
Expression: Enqueue($\mathfrak{H}$(⟨*v*, *i*, *j*⟩), $\mathfrak{H}$(*e*))
Using Rule: (2), Protocol 3

$\mathfrak{H}$(ENQUEUE(⟨*v*, *i*, *j*⟩, *e*)) ≡ $\mathfrak{H}$(⟨Assign(*v*, *e*, *j*), *i*, *j*+1⟩)

It should be pointed that the addition of protocols (2) and (3) does not enhance the generality of the original definition of **expand**. In other words, we can show the following:

Suppose β can be obtained from α in a finite number of expansion steps using a rewriting system **R** under protocols (1), (2) and (3). Then, β can also be obtained from α in a finite number of expansion steps using only protocol (1), provided **R** contains the following rules that specify **if_then_else**:

if_then_else(True, v_1, v_2) $\rightarrow$ v_1

if_then_else(False, v_1, v_2) $\rightarrow$ v_2

The reason for introducing protocols (2) and (3) is to reduce the number of expansion steps needed in the generation of theorems. The two rules of **if_then_else** given above make expansion uneconomical because the right hand side of each of them is a variable. This makes each of them a candidate for being used for expansion at every step of the theorem generation process. Use of protocols (2) and (3) in effect limits the use of the above two rules to cases where there is a rewrite rule with an **if_then_else** in its right hand side, and which could be used for further expansion.

5.3.2 More on the Temporary World

5.3.2.1 The Purpose of TW

The Temporary World (**TW**) serves two purposes: Firstly, it holds information about the invariant $\mathfrak{I}$. Secondly, it provides a means of keeping a log of certain assertions that are needed for temporary stretches during the course of the derivation of an preliminary implementation. Some of these assertions are generated automatically by the procedure; others are supplied by the user.

The information about $\mathfrak{I}$ and the assertions are entered into **TW** as rewrite rules. (The derivation procedure may use the rules in **TW** for expansion like the rules of **PW**, the Perturbed World.) The assertions needed may change during the course of the derivation of a preliminary implementation. Some of the assertions needed can only be determined during the course of the derivation. Because of these reasons, **TW** is treated as a dynamic world, i.e., a world that changes during the course of the derivation of a preliminary implementation. In contrast, **PW** keeps a log of the facts needed through the derivation of the entire preliminary implementation.

There are three reasons why temporary assertions might be needed during the derivation. Firstly, the equation $\mathfrak{H}(\mathbf{F}(\mathbf{g}_1, \ldots, \mathbf{g}_n)) \equiv \mathfrak{H}(?\mathbf{t})$ being searched for is a theorem of PW only under the hypothesis that the arguments to F satisfy $\mathfrak{I}$. The second reason arises in checking if **?rhs** satisfies $\mathfrak{I}$, i.e., if $\mathfrak{I}(?\mathbf{rhs}) \equiv \mathbf{True}$ is a theorem. This check has to be performed under the hypothesis that the arguments to F satisfy $\mathfrak{I}$. Also, performing this check may need the use of the inductive logic. In such a case, it is necessary to set up appropriate hypotheses for the induction.

The third reason for the need for assertions arises while one is attempting to expand a subexpression of a conditional expression **if_then_else(b, $\mathbf{e}_1$, $\mathbf{e}_2$)**. Under such a situation, we may assume that **b** is **False** while expanding a subexpression in the **else**-arm, or that **b** is **True** while expanding a subexpression in the **then**-arm. For instance, consider the expression **if_then_else**($i=j+1$, $\mathbf{e}_2$, **Enqueue**($\mathfrak{H}(\langle v, i, j\rangle)$, $\mathfrak{H}(\mathbf{e}_1)$)). In this case, the subexpression **Enqueue**($\mathfrak{H}(\langle v, i, j\rangle)$, $\mathfrak{H}(\mathbf{e}_1)$) is expandible by the rewrite rule

$\mathfrak{H}(\langle$**Assign(v, e, j), i, j+1**$\rangle) \rightarrow$ **if i = j+1 then Nullq else Enqueue**($\mathfrak{H}(\langle$**v, i, j**$\rangle)$, $\mathfrak{H}$**(e)**)

only if we make the hypothesis that $i = j+1 \rightarrow^*$ **False**.

5.3.2.2 Construction of TW

TW consists of two parts: A *static part*, and a *dynamic part*. The static part remains unchanged for the entire duration of the derivation of the preliminary implementation. The dynamic part may change during the derivation.

5.3.2.2.1 The Static Part

The static part consists of information about the invariant $\mathfrak{I}$. It consists of

(1) A set of rewrite rules that constitute the specification of $\mathfrak{I}$. The specification of $\mathfrak{I}$ involves other data types which are among the implementing types. We assume that the static part contains their specifications also. In the examples we discuss, only the relevant rules from these specifications are displayed.

(2) A set of rewrite rules that express additional properties about $\mathfrak{I}$.

The rewrite rules mentioned in (1), above, can be constructed automatically from the association specification. The information in (2) is something the user has the option of supplying additionally for deriving a preliminary implementation in the presence of a nontrivial invariant. This information is needed for the following reason: There are several preliminary implementations whose derivation is dependent on lemmas that express interesting properties about the invariant. Although it might be possible to prove these lemmas from the specification of $\mathfrak{I}$, the derivation procedure cannot automatically discover the desired lemma. The rewrite rules in (2) specify these lemmas.

The static part of **TW** used for the current example is given below. Rules (1) and (2) are constructed from the specification of $\mathfrak{I}$ given as part of the association specification in Fig. 15. Notice that the right hand side of rule (2) is a simplified version of the right hand side of the corresponding equation of the specification of $\mathfrak{I}$. The rules used in the simplification are (10), (11), (8), and (4). Rule (3) specifies a property of $\mathfrak{I}$. It asserts that if a triple $\langle v, i, j\rangle$ satisfies $\mathfrak{I}$, then $i \leq j$. The property can be proved from the specification of $\mathfrak{I}$ using the KB-method. Rules (4) through (11) belong to the specification **Integer** and **Bool**. These rules will be used in the examples that follow.

(1) $\mathfrak{I}(\langle v, i, i\rangle) \rightarrow \mathbf{True}$

(2) $\mathfrak{I}(\langle \mathbf{Assign}(v, e, j), i, j+1\rangle) \rightarrow i \leq j+1 \wedge [i = j+1 \vee \mathfrak{I}(\langle v, i, j\rangle)]$

(3) $\mathfrak{I}(\langle v, i, j\rangle) \Rightarrow i \leq j \rightarrow \mathbf{True}$

(4) $x = y \vee x \leq y \rightarrow x \leq y$

(5) $\mathbf{True} \vee x \rightarrow \mathbf{True}$

(6) $\sim x \vee x \rightarrow \mathbf{True}$

(7) $\sim(x \wedge y) \rightarrow \sim x \vee \sim y$

(8) $x \vee (y \wedge z) \rightarrow (x \vee y) \wedge (x \vee z)$

(9) $(x \wedge y) \Rightarrow y \rightarrow \mathbf{True}$

(10) $\mathbf{if\_then\_else}(b, \mathbf{True}, e_2) \rightarrow b \vee e_2$

(11) $\mathbf{if\_then\_else}(b, e_1, \mathbf{False}) \rightarrow b \wedge e_1$

5.3.2.2.2 The Dynamic part

This is the part that may change during the course of the derivation of a preliminary implementation. It may vary from the derivation of one rule of the preliminary implementation to another; within the derivation of a single rule, it may vary from one theorem generation step to the next. By a theorem generation step, we mean the following: Recall that the derivation of a rule involves generating a series of theorems. The generation of every theorem in the series is considered as a theorem generation step in the derivation of the rule.

The dynamic part is empty at the beginning of the derivation of every rule of the implementation derfinition. Assertions (in the form of rewrite rules) are added to and removed from the dynamic part at specific instants during the derivation of a rule. Every assertion that is added during the derivation of a rule is removed by the end of the derivation. Every time an assertion is added to **TW**, it is important to ascertain that the addition does not render **TW** inconsistent. To ensure consistency, we run the predicate **Is-an-inductive-theorem-of**[22] (see sec.4.4.3.2) on **TW** every time an assertion is added to **TW**. (Note that **TW** is convergent to begin with. This is because the static part, which consists of the specification of $\mathfrak{I}$, is guaranteed to be convergent.) The assertion is added only if the **Is-an-inductive-theorem-of** succeeds. In some cases the **is-an-inductive-theorem-of** may succeed by generating a finite number of new assertions. In several situations it is useful to add these new assertions also to **TW**. If these assertions are, indeed, added to **TW**, then they should also be removed along with the original assertion.

The assertions in the dynamic part can be classified into two categories based on the life time of their existence. We describe the construction of the two categories below.

Arguments-Assertions

These assertions are added at the beginning of the derivation of a rule. They remain

---

22. We assume that the predicate **Is-an-inductive-theorem-of** is run iteratively a fixed number of times that is finite.

in TW until the end of the derivation of the rule. We call these assertions Arguments-Assertions because they are dependent on the expressions supplied as arguments to the implementing function for which the rule is being derived. For instance, if the rule being derived is of the form $F(g_1, \ldots, g_n) \rightarrow ?t$, then the assertions are dependent on $g_1, \ldots, g_n$.

Arguments-Assertions can be of two kinds: The first kind assert that $g_1, \ldots, g_n$ satisfy $\mathfrak{I}$. These are entered in TW as the rewrite rules $\mathfrak{I}(g_1) \rightarrow \textbf{True}, \ldots, \mathfrak{I}(g_n) \rightarrow \textbf{True}$. It is easy to see that these assertions can be constructed automatically.

The second kind consist of assertions that are supplied by the user. These are used for ensuring that every rule of the preliminary implementation preserves the invariant $\mathfrak{I}$, i.e., $\mathfrak{I}(g_1) \wedge \ldots \wedge \mathfrak{I}(g_n) \Rightarrow \mathfrak{I}(F(g_1, \ldots, g_n))$. The assertions express the induction hypotheses that might be needed for checking the above property. The reason that the user might have to supply these assertion is the following. Recall that our method ensures the invariance property by deriving every rule $F(g_1, \ldots, g_n) \rightarrow ?t$ so that $\mathfrak{I}(?\textbf{rhs}) \equiv \textbf{True}$ is a theorem of TW. (Note that TW already includes rewrite rules asserting that $g_1, \ldots, g_n$ satisfy $\mathfrak{I}$.) If the preliminary implementation desired is such that $\mathfrak{I}(?t) \equiv \textbf{True}$ can be proved automatically from TW using the equational logic or the KB-method for proving inductive properties, then no additional assertions are needed. However, if the preliminary implementation desired is such that the proof of $\mathfrak{I}(?\textbf{rhs}) \equiv \textbf{True}$ needs induction hypotheses that cannot be generated automatically by the KB-method, then assertions expressing the induction hypotheses have to be added to TW.

The assertions used as induction hypotheses in all our examples are constructed by invoking the inference rule given below. The inference rule expresses a general induction principle that uses the termination ordering $\succ$ as the well-founded partial ordering for the induction. Informally, the inference rule can be stated as follows. Suppose $F(g_1, \ldots, g_n) \rightarrow ?t$ is the rule being derived. Then, in trying to ensure $\mathfrak{I}(F(g_1, \ldots, g_n))$, we may assume $\mathfrak{I}(F(v_1, \ldots, v_k))$ for any argument $\langle v_1, \ldots, v_k \rangle$ that satisfies $\mathfrak{I}$, and that is "less than" $\langle g_1, \ldots, g_n \rangle$ in the ordering $\succ$.

$$\frac{\langle g_1, \ldots, g_n \rangle \succ \langle v_1, \ldots, v_k \rangle}{\mathfrak{I}(v_1) \wedge \ldots \wedge \mathfrak{I}(v_k) \Rightarrow \mathfrak{I}(F(v_1, \ldots, v_k))}$$

As an illustration, let us construct a set of Arguments-Assertions for the derivation of a rule for **APPEND**. We will be using these assertions later when we illustrate the derivation of the preliminary implementation for **APPEND**. Suppose we are attempting to derive a rule of the following form:

APPEND($\langle v_1, i_1, j_1\rangle$, $\langle$**Assign**$(v_2, e, j_2), i_2, j_2+1\rangle$) $\rightarrow$ **?rhs**

Then, the Arguments-Assertions may include the following rewrite rules. The first two assertions state that the arguments supplied to **APPEND** satisfy $\mathfrak{I}$. The third assertion is used as an induction hypothesis.

$$\mathfrak{I}(\langle v_1, i_1, j_1\rangle) \rightarrow \textbf{True}$$

$$\mathfrak{I}(\langle \textbf{Assign}(v_2, e, j_2), i_2, j_2+1\rangle) \rightarrow \textbf{True}$$

$$\mathfrak{I}(\langle v_2, i_2, j_2\rangle) \Rightarrow \mathfrak{I}(\textbf{APPEND}(\langle v_1, i_1, j_1\rangle, \langle v_2, i_2, j_2\rangle)) \rightarrow \textbf{True}$$

Conditional-Expressions-Assertions

The second category of assertions in the dynamic part is the Conditional-Expressions-Assertions. A need for these assertions arises while expanding a subexpression of a conditional expression in the generation of theorems. These assertions are added to **TW** at the beginning of a theorem generation step, and removed at the end of the step. The Conditional-Expressions-Assertions needed in a step are determined by the occurrence of the subexpression that is chosen to be expanded for generating the theorem in that step. For instance, suppose the following is the theorem generated in the first step during the derivation of a rule for **APPEND**.

$\mathfrak{H}$(**APPEND**($\langle v_1, i_1, j_1\rangle$, $\langle$**Assign**$(v_2, e, j_2), i_2, j_2+1\rangle$))

$$\equiv \textbf{if\_then\_else}(i_2 = j_2+1, \textbf{Enqueue}(\mathfrak{H}(\textbf{APPEND}(\langle v_1, i_1, j_1\rangle, \langle v_2, i_2, j_2\rangle)), e))$$

Suppose we decide to generate the theorem in step (2) by expanding the subexpression $\mathfrak{H}$(**APPEND**($\langle v_1, i_1, j_1\rangle$, $\langle v_2, i_2, j_2\rangle$)) on the right hand side of the theorem in step (1). Then, we may add to **TW** the assertion $i_2 = j_2+1 \rightarrow$ **False**. The reasoning behind the addition of this assertion should be apparent by now. The subexpression chosen for expansion appears in the **else**-arm of a conditional expression. Hence, while expanding the subexpression we may (if we wish) assume that the corresponding boolean expression is **False**. In general, we

may have to add more than one such assertion in a step because the subexpression could be embedded within more than one conditional expression. Suppose α is the subexpression chosen to be expanded. Then, the Conditional-Expressions-Assertions for the step are determined as follows:

(i) For every conditional expression **if_then_else**(b_1, ...α..., ...), of whose **then**-arm α is a part, add $b_1 \rightarrow$ **True**.

(ii) For every conditional expression **if_then_else**(b_2, ..., ...α...), of whose **else**-arm α is a part, add $b_2 \rightarrow$ **False**.

5.3.3 Preliminary Implementation of Append

This section derives a pair of synthesis equations corresponding to the two rewrite rules in the partial preliminary implementation that define **APPEND**. It illustrates a more interesting utilization of the invariant $\mathfrak{I}$ than was seen in the derivation of the rule for **ENQUEUE**. The derivation also demonstrates how a **where** construct can be introduced into a preliminary implementation, and why it is useful to do so.

Recall the reason for introducing the **where** construct into the preliminary implementation language: To alleviate the limitation of the constraint that a preliminary implementation may not contain any helping functions or observers of the representation type. The constraint, in particular, makes it impossible to select the components of a tuple returned by an expression that appears on the right hand side of a rule.

For instance, suppose we wish to construct a triple using the components of the triple returned by **APPEND**($\langle v_1, i_1, j_1\rangle$, $\langle v_2, i_2, j_2\rangle$). A **where** construct permits us to do this by rewriting the above expression in the following fashion.

$\langle v, i, j\rangle$ **where** $\langle v, i, j\rangle$ **is** **APPEND**($\langle v_1, i_1, j_1\rangle$, $\langle v_2, i_2, j_2\rangle$)

Then, the first argument can be further transformed to construct the desired triple. For instance,

Assign(v, e_2, j), $i, j+1\rangle$ **where** $\langle v, i, j\rangle$ **is** **APPEND**($\langle v_1, i_1, j_1\rangle$, $\langle v_2, i_2, j_2\rangle$)

The new terminals v, i, j introduced should be distinct from the terminals that

already exist in the expression that is being transformed. It should be noted that a **where** construct can always be eliminated from an expression provided we are permitted to use the selector operations of the tuple type. This elimination can also be performed automatically. For instance, the **where** construct in the above expressions can be eliminated by systematically replacing every occurrence of v, i, and j in the first argument to the **where** construct by the following expressions: **First(APPEND(**$\langle v_1, i_1, j_1\rangle$, $\langle v_2, i_2, j_2\rangle$**))**, **Second(APPEND(**$\langle v_1, i_1, j_1\rangle$, $\langle v_2, i_2, j_2\rangle$**))**, **Third(APPEND(**$\langle v_1, i_1, j_1\rangle$, $\langle v_2, i_2, j_2\rangle$**))**. (**First**, **Second**, and **Third** are operations that select the first, second, and third components of a triple.)

Below, we give two rules concerning a **where** construct. The rules can be used at any step during the generation of theorems to transform the expression on the right hand side of a theorem. The first rule specifies how a **where** construct can be introduced into an expression. The second rule specifies how the position of **where** can be moved within an expression without altering its semantics. Suppose

(1) F is an implementing function whose range is a triple type,

(2) g is an arbitrary function,

(3) $e, e_1, \ldots, e_k$ are arbitrary expressions,

(4) v, i, j are terminals that do not appear in the equation $e \equiv g(..., F(e_1, \ldots, e_k), ...)$.

Where-Rule (1)

$$\frac{e \equiv g(..., F(e_1, \ldots, e_k), ...)}{e \equiv g(..., \langle v, i, j\rangle \text{ where } \langle v, i, j\rangle \text{ is } F(e_1, \ldots, e_k), ...)}$$

Where-Rule (2)

$$\frac{e \equiv g(..., \langle v, i, j\rangle \text{ where } \langle v, i, j\rangle \text{ is } F(e_1, \ldots, e_k), ...)}{e \equiv g(..., \langle v, i, j\rangle, ...) \text{ where } \langle v, i, j\rangle \text{ is } F(e_1, \ldots, e_k)}$$

A few remarks are in order at this point regarding expanding an expression that appears as a subexpression of a **where** construct. Firstly, an instance of a **where** construct is treated, for syntactic purposes, as an application of a function **Where_is** with three arguments. For instance, **<Assign(**v, e_2, j**), $i, j+1$> where <**v, i, j**> is APPEND(**$\langle v_1, i_1, j_1\rangle$, $\langle v_2, i_2, j_2\rangle$**)** is

treated as the expression **Where_is**(<**Assign**(v, e_2, j), i, $j+1$>, <v, i, j>, **APPEND**(<v_1, i_1, j_1>, <v_2, i_2, j_2>)). Secondly, the second argument to **Where_is** may not be expanded; only, the first and the third may be expanded. In the above example, for instance, <v, i, j> may not be expanded. This is because the second argument to **Where_is** has to be a tuple of terminals (or variables). It does not make sense to have a nonterminal expression as a part of the second argument; expansion will introduce a nonterminal expression.

The third remark concerns the possibility of making temporary assertions while expanding subexpressions of a **where** expression. Consider the example given above. Suppose we decide on expanding the expression <**Assign**(v, e_2, j), i, $j+1$>. The terminals v, i, and j in this expression are such that <v, i, j> is acting as a place holder for **APPEND**(<v_1, i_1, j_1>, <v_2, i_2, j_2>). If **APPEND**(<v_1, i_1, j_1>, <v_2, i_2, j_2>) happens to be such that $\mathfrak{I}$(**APPEND**(<v_1, i_1, j_1>, <v_2, i_2, j_2>)) is **True**, then we may assume that $\mathfrak{I}$(<v, i, j>) is also **True** as long we are expanding the first argument to the **where** expression. This assumption may, in general, enhance the possibility for expansion. Thus, expansion of a subexpression of a **where** expression may result in an update of the Temporary World (**TW**). For instance, in the above example, if we $\mathfrak{I}$(**APPEND**(<v_1, i_1, j_1>, <v_2, i_2, j_2>)) ≡ **True** is a theorem of **TW**, then we may update **TW** with the assertion $\mathfrak{I}$(<v, i, j>) → **True**. This is used in the derivation to follow.

ArgsSet = { **Arg1**: <<v_1, i_1, j_1>, <v_2, i_2, i_2>>
Arg2: <<v_1, i_1, j_1>, <**Assign**(v_2, e_2, j_2), i_2, j_2+1>> }

Relevant Excerpts of the Perturbed World

(1) $\mathfrak{H}$(<v, i, i>) → **Nullq**
(2) $\mathfrak{H}$(<**Assign**(v, e, j), i, j+1>) → **if_then_else**(i = j+1, **Nullq**, **Enqueue**($\mathfrak{H}$(<v, i, j>), $\mathfrak{H}$(e)))

(3) $\mathfrak{H}$(**APPEND**(x, y)) → **Append**($\mathfrak{H}$(x), $\mathfrak{H}$(y))
(4) $\mathfrak{H}$(**if_then_else**(b, v_1, v_2)) → **if_then_else**(b, $\mathfrak{H}$(v_1), $\mathfrak{H}$(v_2))

Derivation of the rule corresponding to Arg1

Form of the theorem to be generated: $\mathfrak{H}$(**APPEND**(<v_1, i_1, j_1>, <v_2, i_2, i_2>)) ≡ $\mathfrak{H}$(?rhs_1)
$\mathfrak{H}$(**APPEND**(<v_1, i_1, j_1>, <v_2, i_2, i_2>))↓: $\mathfrak{H}$(<v_1, i_1, j_1>)
Rules used for simplification:

Initial State of the Temporary World

$\mathfrak{I}(\langle v, i, i\rangle) \rightarrow$ True

$\mathfrak{I}(\langle \text{Assign}(v, e, j), i, j+1\rangle) \rightarrow i \leq j+1 \wedge [i = j+1 \vee \mathfrak{I}(\langle v, i, j\rangle)]$

$\mathfrak{I}(\langle v, i, j\rangle) \Rightarrow i \leq j \rightarrow$ True

$\mathfrak{I}(\langle v_1, i_1, j_1\rangle) \rightarrow$ True

---

Step (1) Invoke Synthesis Rule (1) on $\mathfrak{IG}(\text{APPEND}(\langle v_1, i_1, j_1\rangle, \langle v_2, i_2, i_2\rangle))$
$\mathfrak{IG}(\text{APPEND}(\langle v_1, i_1, j_1\rangle, \langle v_2, i_2, i_2\rangle)) \equiv \mathfrak{IG}(\langle v_1, i_1, j_1\rangle)$

$\text{APPEND}(\langle v_1, i_1, j_1\rangle, \langle v_2, i_2, i_2\rangle) \rightarrow \langle v_1, i_1, j_1\rangle$

Derivation of the rule corresponding to Arg2

Form of the theorem to be generated: $\mathfrak{IG}(\text{APPEND}(\langle v_1, i_1, j_1\rangle, \langle \text{Assign}(v_2, e_2, j_2), i_2, j_2+1\rangle)) \equiv \mathfrak{IG}(?\text{rhs}_2)$
$\mathfrak{IG}(\text{APPEND}(\langle v_1, i_1, j_1\rangle, \langle \text{Assign}(v_2, e_2, j_2), i_2, j_2+1\rangle))\downarrow$:

$\text{if\_then\_else}(i_2 = j_2+1, \mathfrak{IG}(\langle v_1, i_1, j_1\rangle),$
$\text{Enqueue}(\text{Append}(\mathfrak{IG}(\langle v_1, i_1, j_1\rangle), \mathfrak{IG}(\langle v_2, i_2, j_2\rangle)), \mathfrak{IG}(e_2)))$

Rules used for simplification:

Initial State of the Temporary World

Static Part

(5) $\mathfrak{I}(\langle v, i, i\rangle) \rightarrow$ True

(6) $\mathfrak{I}(\langle \text{Assign}(v, e, j), i, j+1\rangle) \rightarrow i \leq j+1 \wedge [i = j+1 \vee \mathfrak{I}(\langle v, i, j\rangle)]$

(7) $[\mathfrak{I}(\langle v, i, j\rangle) \Rightarrow i \leq j] \rightarrow$ True

Arguments-Assertions

(8) $\mathfrak{I}(\langle v_1, i_1, j_1\rangle) \rightarrow$ True

(9) $\mathfrak{I}(\langle \text{Assign}(v_2, e_2, j_2), i_2, j_2+1\rangle) \rightarrow$ True

("The following is as a consequence of Rule (9)")

(9a) $i_2 \leq j_2+1 \wedge [i_2 = j_2+1 \vee \mathfrak{I}(\langle v_2, i_2, j_2\rangle)] \rightarrow$ True

(10) $\mathfrak{I}(\langle v_2, i_2, j_2\rangle) \Rightarrow \mathfrak{I}(\text{APPEND}(\langle v_1, i_1, j_1\rangle, \langle v_2, i_2, j_2\rangle)) \rightarrow$ True

---

Step (1) Invoke Synthesis Rule (1) on the expression $\mathcal{H}$(APPEND(<v_1, i_1, j_1>, <v_2, i_2, i_2>))

$\mathcal{H}$(APPEND(<v_1, i_1, j_1>, <v_2, i_2, i_2>)) ≡

if_then_else($i_2 = j_2 + 1$, $\mathcal{H}$(<v_1, i_1, j_1>),

Enqueue(Append($\mathcal{H}$(<v_1, i_1, j_1>), $\mathcal{H}$(<v_2, i_2, j_2>)), $\mathcal{H}$(e_2)))

Step (2) Expand: Occurrence: 3.1

Expression: Append($\mathcal{H}$(<v_1, i_1, j_1>), $\mathcal{H}$(<v_2, i_2, j_2>))

Using Rule: (3)

$\mathcal{H}$(APPEND(<v_1, i_1, j_1>, <Assign(v_2, e_2, j_2), i_2, j_2+1>)) ≡

if_then_else($i_2 = j_2 + 1$, $\mathcal{H}$(<v_1, i_1, j_1>),

Enqueue($\mathcal{H}$(APPEND(<v_1, i_1, j_1>, <v_2, i_2, j_2>)), $\mathcal{H}$(e_2)))

Step (3) Transform: Occurrence: 3.1.1

Expression: APPEND(<v_1, i_1, j_1>, <v_2, i_2, j_2>)

Using Rule: where-rule (1)

$\mathcal{H}$(APPEND(<v_1, i_1, j_1>, <Assign(v_2, e_2, j_2), i_2, j_2+1>)) ≡

if_then_else($i_2 = j_2 + 1$, $\mathcal{H}$(<v_1, i_1, j_1>), Enqueue($\mathcal{H}$(<v, i, j>), $\mathcal{H}$(e_2))

where <v, i, j> is APPEND(<v_1, i_1, j_1>, <v_2, i_2, j_2>))

Step (4) Expand: Occurrence: 3

Expression: Enqueue($\mathcal{H}$(<v, i, j>), $\mathcal{H}$(e_2))

Using Rule:

TW Update:

Added because expression is in scope of **else-arm**

$i_2 = j_2 + 1$ **False**

$i_2 \leq j_2 + 1 \wedge \mathcal{I}$(<$v_2$, i_2, j_2>) → **True**

$\mathcal{I}$(<v_2, i_2, j_2>) → **True**

$\mathcal{I}$(APPEND(<v_1, i_1, j_1>, <v_2, i_2, j_2>)) → **True**

Added because expression is in scope of **where**

$\mathcal{I}$(<v, i, j>) → **True**

$i \leq j$ **True**

$\mathcal{H}$(APPEND(<v_1, i_1, j_1>, <Assign(v_2, e_2, j_2), i_2, j_2+1>)) ≡

if_then_else($i_2 = j_2 + 1$, $\mathcal{H}$(<v_1, i_1, j_1>), $\mathcal{H}$(<Assign(v, e_2, j), i, $j+1$>)

where $\langle v, i, j\rangle$ is APPEND($\langle v_1, i_1, j_1\rangle, \langle v_2, i_2, j_2\rangle$))

Step (5) Transform: Occurrence:

Using Rule: where-rule (2)

$\mathfrak{G}$(APPEND($\langle v_1, i_1, j_1\rangle, \langle$Assign($v_2, e_2, j_2$), $i_2, j_2+1\rangle$)) ≡

if_then_else($i_2 = j_2+1$, $\mathfrak{G}(\langle v_1, i_1, j_1\rangle)$, $\mathfrak{G}(\langle$Assign(v, e_2, j), $i, j+1\rangle$))

where $\langle v, i, j\rangle$ is APPEND($\langle v_1, i_1, j_1\rangle, \langle v_2, i_2, j_2\rangle$)

Step(6) Expand: Occurrence: λ

Using Rule: (4)

$\mathfrak{G}$(APPEND($\langle v_1, i_1, j_1\rangle, \langle$Assign($v_2, e_2, j_2$), $i_2, j_2+1\rangle$)) ≡

$\mathfrak{G}$(if_then_else($i_2 = j_2+1, \langle v_1, i_1, j_1\rangle, \langle$Assign($v, e_2, j$), $i, j+1\rangle$))

where $\langle v, i, j\rangle$ is APPEND($\langle v_1, i_1, j_1\rangle, \langle v_2, i_2, j_2\rangle$)

APPEND($\langle v_1, i_1, j_1\rangle, \langle$Assign($v_2, e_2, j_2$), $i_2, j_2+1\rangle$) →

if $i_2 = j_2+1$ then $\langle v_1, i_1, j_1\rangle$

else $\langle$Assign(v, e_2, j), $i, j+1\rangle$ where $\langle v, i, j\rangle$ is APPEND($\langle v_1, i_1, j_1\rangle, \langle v_2, i_2, j_2\rangle$)

Definition of APPEND

APPEND($\langle v_1, i_1, j_1\rangle, \langle v_2, i_2, i_2\rangle$) → $\langle v_1, i_1, j_1\rangle$

APPEND($\langle v_1, i_1, j_1\rangle, \langle$Assign($v_2, e_2, j_2$), $i_2, j_2+1\rangle$) →

if $i_2 = j_2+1$ then $\langle v_1, i_1, j_1\rangle$

else $\langle$Assign(v, e_2, j), $i, j+1\rangle$ where $\langle v, i, j\rangle$ is APPEND($\langle v_1, i_1, j_1\rangle, \langle v_2, i_2, j_2\rangle$)

6. Stage 2: The Target Implementation

The second stage of the synthesis procedure transforms the preliminary implementation of the implemented type into a target implementation. For instance, in the example implementing **Queue_Int** in terms of **Circ_List**, the preliminary implementation derived in the last chapter (shown Fig. 5 of chapter 2) is transformed into a target implementation such as the one shown in Fig. 0.

There are two differences between a preliminary implementation and a target implementation. The first one is that in a preliminary implementation the only operations of the representation type allowed to appear are the generators of the type. The target implementation may also contain nongenerators of the type. The second difference is in the function definition methods used by the two forms of implementation. In a preliminary implementation a function is defined by means of a set of rewrite rules. For example the preliminary implementation of ENQUEUE (Fig. 5) is:

ENQUEUE(Create, j) → Insert(Create, j)

ENQUEUE(Insert(c, i), j) → Insert(ENQUEUE(c, j), i)

In a target implementation a function is defined by means of a single expression. For example, ENQUEUE is defined as: **ENQUEUE(d, k) :: = Rotate(Insert(d, k))**. The transformation performed takes into consideration both of these differences.

It should be noted that a preliminary implementation is an executable

---

Fig. 18. An Implementation

```
NULLQ() :: = Create()

ENQUEUE(c, j) :: = Rotate(Insert(c, j))

FRONT(c) :: = Value(c)

DEQUEUE(c) :: = Remove(c)

APPEND(c, d) :: = Join(d, c)

SIZE(c) :: = if Empty(c) then 0
             else SIZE(Remove(c)) + 1
```

implementation. It can be executed by an interpreter that simplifies algebraic expressions using the rewrite rules in the preliminary implementation and the specifications of the implementing types. The interpreter must have a pattern matching capability to invoke the appropriate rewrite rule while simplifying an expression. The program verification system AFFIRM [39], and the programming system PROLOG [??] provide such an interpreter. Given the specifications of all the implementing types, the interpreter can execute the preliminary implementation on any given input. For example, the value returned by the operation (of **Queue_Int**) **Front** on the queue constructed by **Enqueue(Nullq, 1)** is obtained by finding the normal form of **FRONT(ENQUEUE(NULLQ( ), 1))** using the preliminary implementation: The normal form is **1**. Depending on the range type of the operation, the normal form can, in general, be a generator constant of any of the implementing types. The normal form can then be evaluated assuming there exist implementations for the implementing types.

Our goal is to derive the target implementation in a form that can be compiled by a compiler for an applicative language. The motivation for this is primarily one of efficiency. There are two reasons why a target implementation is more efficient than a preliminary implementation. The first one arises because of the freedom to use nongenerators of the representation type in a target implementation. This enables one, in some instances, to eliminate recursion from the preliminary implementation of an operation, and to transform it into a target implementation which is merely a composition of the operations of the implementing types. The implementation of **ENQUEUE** shown above is an instance of such a situation. The use of the operation **Rotate** in the target implementation eliminates the recursion which was essential in the preliminary implementation. The second reason is that an implementation that can be compiled by means of a conventional compiler is in general more efficient than interpreting a set of rewrite rules.

We develop two methods of deriving a target implementation from a preliminary implementation: The Recursion Preserving Method, and the Recursion Eliminating Method. Both the methods are based upon expansion using rewrite rules. The target implementations derived by the first method preserve any recursion that may exist in the corresponding preliminary implementations. The second method can eliminate recursion from a

preliminary implementation of an operation if there exists a nonrecursive implementation for the operation. The second method is more general because it can also derive the implementations derived by the first method. The advantage of the first method is that it is, in general, faster than the second in situations where the two methods derive the same target implementation.

6.1 The Recursion Preserving Method

This method uses a special set of functions, called the *inverting functions*, on the implementing types for transforming a preliminary implementation into a target implementation. To understand what inverting functions are and how they are useful in deriving a target implementation, let us take a closer look at the difference in the function definition methods used by the two forms of implementation. The preliminary implementation for **SIZE** is

SIZE(Create) → 0

SIZE(Insert(c, i)) → SIZE(c) + 1,

and a possible target implementation for it is

SIZE(d) ::= if Empty(d) then 0

else SIZE(Remove(d)) + 1.

In the preliminary implementation, the argument to **SIZE** on the left hand side of a rule may be a generator expression. The argument indicates the structure of the expression that constructs the values for which the rewrite rule is applicable. This freedom serves two purposes in a preliminary implementation. Firstly, it is used for performing a case analysis based on the structure of the argument. Secondly, the explicit indication of the structure of the arguments on the left hand side makes the decomposition of the arguments trivial. For instance, in the second rewrite rule for **SIZE** the variable **c** used on the right hand side is actually a component of the argument to **SIZE**. We were able to access this component without actually having to generate code to decompose the argument.

In a target implementation, the argument to **SIZE** on the left hand side of the

definition is a variable. This means that the expression on the right hand side of the definition must have explicit pieces of "code" to perform the case analysis based on the structure of the argument, and to decompose the argument. For instance, in the target implementation of **SIZE** given above, the subexpression **Remove(d)** extracts the component of the argument **d** that is denoted by the variable **c** in the preliminary implementation. The subexpression **Empty(d)** checks if **d** is a value constructed by **Create**; the **if_then_else** expression performs the desired case analysis. Let us call the subexpressions that perform these functions mentioned above *inverting expressions.*

A preliminary implementation can be systematically transformed into a target implementation if the inverting expressions can be generated automatically. The inverting functions of the implementing types serve precisely this purpose. For instance, in the above example **Remove** and **Empty** are two of the inverting functions for **Circ_List**. The inverting expressions can be automatically derived in terms of the inverting functions. Thus, the transformation of a preliminary implementation into a target implementation according to this method consists of two steps: First, determine the inverting expressions in terms of the inverting functions; second, derive implementations for the inverting functions in terms of the operations of the implementing types. The two subsections to follow describe the two steps.

6.1.1 Inverting Functions and Inverting Expressions

Inverting functions[23] of a data type are a family of functions on the type that are inter-related in a special way. Inverting functions are defined with respect to a basis of the type. The relationship among the inverting functions of a family is such that the functions can be used to algorithmically invert the process of constructing a value from the generators of the type. In other words, it is possible to construct algorithmically the inverting

---

23. Inverting functions are related to *distinguished functions* defined in [24]. A family of inverting functions for a data type can also serve as a family of distinguished functions. The reverse implication is not true in general. In [24] distinguished functions are used to formalize the expressive power of a data type.

expressions as a composition of appropriate inverting functions. The inverting expressions perform the following functions:

(1) Given a variable **v** and a generator expression **t**, check if the value denoted by **v** can be constructed by a generator expression that has the form of **t**. Since an inverting expression that performs this function is normally a boolean expression, we call it a *boolean inverting* expression.

(2) Assuming that a given variable **v** denotes a value that is constructed by an expression that has the form of a given generator expression **t**, determine the various components of **t** from **v**. We call an inverting expression that performs this function a *component inverting* expression since it extracts a component of a generator expression.

For example, the operations **Remove**, **Value**, and **~(Empty)** can serve as a family of inverting functions for **Circ_List**. This is because the inverting expressions for any generator expression of **Circ_List** can be automatically constructed from these operations. For instance, suppose **v** is a variable of type **Circ_List**, and **t = Insert(Insert(c, i), j)** is the generator expression under consideration. The following are some of the inverting expressions for **t**:

(1) **Not(Empty(Remove(v)))** is a boolean inverting expression for **t**. It checks if **v** denotes a value constructed by a generator expression that has the form of **t**.

(2) Some of the component inverting expressions of **t** are **Value(v)** which extracts **j**, **Remove(Remove(v))** which extracts **c**, and **Value(Remove(v))** which extracts **i**.

Let us now formalize the properties that characterize a family of inverting functions for an arbitrary data type. We express the properties in the form of rewrite rules. The properties are such that they do not necessarily characterize a unique set of functions. This is done deliberately to offer flexibility in choosing an implementation for the inverting functions. Inverting functions are always defined with respect to a basis for the data type. Let the basis for the data type be $\mathfrak{B} = \{\sigma_i \mid i>0\}$. Inverting functions can be classified into two categories: the *component inverting* functions and the *boolean inverting* functions.

(1) There is a set of **n** component inverting functions $(\mathbf{d}_1, \ldots, \mathbf{d}_n)$ associated with every generator σ_i in the basis whose arity is **n**. They are characterized by the following property:

$\sigma_i(\mathbf{d}_1(\sigma_i(\mathbf{v}_1, \ldots, \mathbf{v}_n)), \ldots, \mathbf{d}_n(\sigma_i(\mathbf{v}_1, \ldots, \mathbf{v}_n))) \rightarrow \sigma_i(\mathbf{v}_1, \ldots, \mathbf{v}_n)$

A generator whose arity is zero does not have any associated component inverting functions. The component inverting functions associated with σ_i factor a value constructed by σ_i. They return the arguments used by σ_i in constructing the value. At the outset it may appear more natural to characterize the component inverting functions as follows: $\mathbf{d}_j(\sigma_j(\mathbf{v}_1, \ldots, \mathbf{v}_n)) \rightarrow \mathbf{v}_j$. The problem with such a characterization is that it may result in ill-defined component inverting functions in situations where the generators can be used in more than one way to construct the same value. For instance, consider the basis $\mathfrak{B} = \{0, 1, +\}$ for **Natural_Numbers.** If $\mathbf{d}_1$ associated with + is defined as $\mathbf{d}_1(x+y) \rightarrow x$, then we have a situation where $\mathbf{d}_1(0+1) = 0$ and $\mathbf{d}_1(1+0) = 1$. This will conflict with the rest of the specification of type **Natural_Numbers** which should allow us to prove that $(0+1) = (1+0)$.

(2) There is a boolean inverting function associated with every generator in the basis. The boolean inverting function, $\mathbf{p}_i$, associated with a generator σ_i returns **True** on values that can be constructed by a generator expression that has the form $\sigma_i(\mathbf{v}_1, \ldots, \mathbf{v}_k)$. So, $\mathbf{p}_i$ is characterized by $\mathbf{p}_i(v) \rightarrow \sigma_i(\mathbf{d}_1(v), \ldots, \mathbf{d}_n(v)) = v$, where = is the equality operation on the type. Thus, the recursion preserving method in general applies only when each of the implementing types has the **equal** operation defined on it. A simpler characterization, which applies only when the basis is such that every value of the type can be constructed uniquely using the generators is as follows:

$\mathbf{p}_i(\sigma_i(\mathbf{v}_1, \ldots, \mathbf{v}_n)) \rightarrow$ **True.**

$\mathbf{p}_i(\sigma_j(\mathbf{v}_1, \ldots, \mathbf{v}_n)) \rightarrow$ **False** $(i \neq j)$

The basis for **Circ_List** is $\mathfrak{B} =$ **{Create, Insert}**. It has two component inverting functions ($\mathbf{d}_1$ and $\mathbf{d}_2$) both of which are associated with **Insert**, and characterized by $\mathbf{Insert}(\mathbf{d}_1(\mathbf{insert}(v, i)), \mathbf{d}_2(\mathbf{Insert}(v, i))) \rightarrow \mathbf{Insert}(v, i)$. It has two boolean inverting functions, $\mathbf{p}_1$

and p_2, one associated with **Create** and the other associated with **Insert**. They are characterized as follows. (Note that the generators of **Circ_List** are such that every circular list can be constructed uniquely in terms of the generators.)

p_1(**Create**) → **True**

p_1(**Insert(c, i)**) → **False**

p_2(**Insert(c, i)**) → **True**

p_2(**Create**) → **False**

Notice that p_1 and p_2, in this case, are complement of each other. So, while deriving implementations for the inverting functions, we implement only p_1; p_2 is obtained as a negation of p_1.

It is not hard to see how a preliminary implementation can be transformed into a target implementation in terms of the inverting functions. Fig. 19 gives a general procedure that does it for an arbitrary preliminary implementation. In the following, we illustrate the procedure on the preliminary implementation of **SIZE**. The preliminary implementation **SIZE** consists of the following rewrite rules.

SIZE(Create) → **0**

SIZE(Insert(c, i)) → **SIZE(c) + 1**

Suppose the left hand side of the target implementation is **SIZE(v)**. The expression on the right hand side is a nested **if_then_else** expression that performs a case analysis. There is a case corresponding to every rewrite rule in the preliminary implementation. In the present case the right hand side would have the following form:

if b_1 **then** e_1

else if b_2 **then** e_2

The expressions b_1 and e_1 are determined from the first rewrite rule using the inverting expressions associated with the generator expression that appears as the argument to **SIZE** on the left hand side of the rewrite rule. The expressions b_2 and e_2 are determined similarly from the second rewrite rule. We will describe how b_2 and e_2 are determined since they are more

Fig. 19. The Procedure RPM

Suppose the prcliminary implementation of F consists of the following rules:

$$F(g_1) \rightarrow t_1$$
$$F(g_2) \rightarrow t_2$$
$$\vdots$$
$$F(g_n) \rightarrow t_n$$

Then, the target implementation for F is

F(v) ::= if b_1 then s_1
else if b_2 then s_2
⋮
else if b_n then s_n

where

(1) b_i is the boolean inverting expression of g_i which is obtained by the procedure **BIE** described below.

(2) s_i is the expression obtained by replacing every terminal in t_i by the component inverting expression of g_i that extracts te terminal. This is obtained by the procedure **CIE** described below.

For convenience, we assume that the generators have an arity that is at most one.

CIE = proc (α: generator expression, u: Occurrence)
returns (component inverting expression)

Suppose α is $\sigma(\alpha_1)$
d is the d-function associated with σ

if $u = \lambda$ then return(λ)
else if $u = 1.v$ then return(d ∘ CIE(α_1, v)

end CIE

BIE = proc (α: generator expression) returns (boolean inverting expression)

if α is a variable then return(λ)
else if $\alpha = \sigma(\alpha_1)$
then return(p ∘ $\wedge$ ∘ BIE1(α_1, d))
where p is the boolean inverting function associated with σ

d is the component inverting function associated with σ

BIE1 = proc (α: genarator expression, d: inverting function symbol)
returns (boolean inverting expression)

if α is a variable then return(λ)
else if $\alpha = \sigma(\alpha_1)$
then return ($p \circ d \circ BIE(\alpha_1)$)
where p is the boolean inverting function associated with σ

interesting than the determination of b_1 and e_1. b_2 is the expression that determines if v denotes a value constructed from an expression that has the form of **Insert(c, i)**, so b_2 is $p_2(v)$. e_2 is identical to **SIZE(c) + 1** except for the following modification: The variables c and i, which denote the components of the expression appearing as argument to **SIZE** on the left hand side of the rule, are replaced by the corresponding inverting expressions that extract those components from v. That is, c is replaced by $d_1(v)$ and i is replaced by $d_2(v)$. So, e_2 is $SIZE(d_1(v)) + 1$. b_1 and e_1 can be determined similarly. So the target implementation for **SIZE** in terms of the inverting functions is below:

$$SIZE(v) ::= \textbf{if } p_1(v) \textbf{ then } 0$$
$$\textbf{else if } p_2(v) \textbf{ then } SIZE(d_1(v)) + 1$$

6.1.2 Implementations for the Inverting Functions

Implementations for the inverting functions are derived using the recursion eliminating method described in the next section. Note that the properties characterizing the inverting functions are expressed by means of a set of rewrite rules. Implementations for the inverting functions are determined by searching for appropriate compositions of the operations of the implementing types that satisfy the rewrite rules characterizing the inverting functions. In the following we show the theorem generation sequences that derive implementations for each of the inverting functions used above.

Derivation for d_1 and d_2

Relevant Rewrite Rules used for Expansion

.....
(1) Value(Create) → ERROR
(2) Value(Insert(c, i)) → i
(3) Remove(Create) → ERROR
(4) Remove(Insert(c, i)) → c
.....

Form of the theorem to be generated: $Insert(v, i) \equiv Insert(f^*_1(Insert(v, i)), f^*_2(Insert(v, i)))$
Normal form of **Insert(v, i)**: **Insert(v, i)**
Rules used for the normal form: None

Step (1) Invoke Synthesis Rule (1) on **0**

Insert(v, i) ≡ Insert(v, i)

Step (2) Expand Expression: **v**

Using Rule: (4)

Insert(v, i) ≡ Insert(Remove(Insert(v, i)), i)

Step (3) Expand Expression: **i**

Using Rule: (2)

Insert(v, i) ≡ Insert(Remove(Insert(v, i)), Value(Insert(v, i)))

The above theorem determines the following solutions for f^*_1 and f^*_2: **Remove** and **Value**. Therefore, we have the following implementations for d_1 and d_2.

$d_1(v) ::=$ **Remove(v)**

$d_2(v) ::=$ **Value(v)**

Derivation for p_1

Relevant Rewrite Rules used for Expansion

.....

(8) Empty(Create) → true

(9) Empty(Insert(c, i)) → false

.....

Form of the theorem to be generated: **True ≡ f*(Create))**

Normal form of **True**: **True**

Rules used for the normal form: None

Step (1) Invoke Synthesis Rule (1) on **True**

True ≡ True

Step (2) Expand Expression: **True**

Using Rule: (8)

True ≡ Empty(Create)

The last theorem determines the following solution for **f***: **Empty**. Note that this function also satisfies the other rewrite rule characterizing p_1, namely p_1(**Insert**(c,i)) → **False**. Therefore, p_1 can be implemented as follows:

p_1(v) ::= **Empty**(v)

6.2 The Recursion Eliminating Method

Let us suppose we are deriving a target implementation for an implementing function F whose preliminary implementation consists of the set of rewrite rules given below.

$F(g_1) \rightarrow t_1$

•

•

•

$F(g_n) \rightarrow t_n$

We assume that F is a single variable function for convenience. The general description of the method given below can be extended easily to a multivariable function. In a target implementation, the function F is defined as F(v) ::= e, where v is a variable, and e is an expression containing v and any of the following function symbols:

(1) Operations of the implementing types

(2) The implementing functions

(3) The function **if_then_else**

Let us denote e as f*(v), where f* is some composition of the function symbols listed above. The derivation of a target implementation consists of finding a suitable f*. The composition f* should be such that the function defined by F(v) ::= f*(v) has the same behavior as the one defined by the set of rewrite rules given above.

To characterize the problem formally, we define the following concept. A composition f* *satisfies* a rewrite rule of F if the equation obtained by substituting f* for F on both the sides of the rewrite rule is a theorem of the rewriting system consisting of the

preliminary implementation and the specifications of the implementing types. For example, the composition **Rotate(Insert(d, k))** satisfies the rule **ENQUEUE(Insert(c, i), j) → Insert(ENQUEUE(c, j), i)** if the equation **Rotate(Insert(Insert(c, i), j)) ≡ Insert(Rotate(Insert(c, j)), i)** is a theorem.

The composition **f*** to be derived should be such that **f*** satisfies each of the rewrite rules in the preliminary implementation of **F**. That is, the following equations should be theorems. (The notation $t_1[F \leftarrow f^*]$ denotes the expression obtained by replacing **F** by **f*** in t_1.)

$$f^*(g_1) \equiv t_1[F \leftarrow f^*]$$

$$\vdots$$

$$f^*(g_n) \equiv t_n[F \leftarrow f^*]$$

The purpose of the above formulation (of the condition that a solution for **f*** is supposed to satisfy) is to allow us to use a theorem generation strategy similar to the one used in deriving a preliminary implementation. We generate a theorem using one of the above equations as a template by treating **f*** as a place holder in the equation. Let us call this equation the *template* equation. A theorem that has the form of the template equation determines a candidate for **f***. A single theorem may determine more than one candidate for **f***, but only finitely many, because the expressions we are dealing with have finite size. The candidate(s) can be determined automatically by comparing the theorem with the template equation. The goal is to generate a theorem that not only has the form of the template equation but is also such that the candidate for **f*** satisfies the rest of the equations in the preliminary implementation of **F**.

The generation of theorems is carried out in the same fashion as in deriving the preliminary implementation. We use the same set of synthesis rules developed earlier. The theorems that are of interest to us in the present situation involve only the operations of the implementing types and the implementing functions. Therefore, the rewriting system that is used for performing expansion (while generating the theorems) consists of the preliminary implementation and the specifications of the implementing types. In contrast, the rewriting system used in the derivation of the preliminary implementation consisted of the

specifications of the implemented type and the association specification. Note that the preliminary implementation did not exist at that time. Checking if a candidate for **f*** satisfies the rewrite rules essentially involves checking if an equation is a theorem.

Let us illustrate the method on the derivation of the target implementation for **ENQUEUE** shown earlier. The preliminary implementation of **ENQUEUE** is repeated below for ease of reference.

ENQUEUE(Create, j) → Insert(Create, j)

ENQUEUE(Insert(c, i), j) → Insert(ENQUEUE(c, j), i)

The **f*** to be derived should be such that the following equations are theorems. (Note that the equations are obtained by replacing **ENQUEUE** by **f*** in the rewrite rules, and then interchanging the two sides. The reason for interchanging the sides will be explained shortly.)

(1) **Insert(Create, j) ≡ f*(Create, j)**

(2) **Insert(f*(c, j), i) ≡ f*(Insert(c, i), j)**

We use equation (1) as the template equation. The nature of our synthesis rules imposes certain restrictions on the equations that can be used as template. The synthesis rules are formulated to generate theorems with a known left hand side, but an unknown right hand side. So, the template equation should be such that the unknown entity **f*** appears only on the right hand side. In equation (2) both sides are unknown since **f*** occurs on both the sides. This was also the reason behind interchanging the two sides of the rewrite rules while obtaining the above equations. Note that there always exists at least one equation with a known right hand side. This corresponds to the rewrite rule in the preliminary implementation of F that represents the basis case.

Shown below is a sequence of steps that generates a theorem that gives rise to a target implementation.

Relevant Rewrite Rules used for Expansion

.....

(3) Rotate(Create) → Create

(4) Rotate(Insert(Create, i)) → Insert(Create, i)

(5) Rotate(Insert(Insert(c, i1), i2)) → Insert(Rotate(Insert(c, i2)), i1)

.....

Form of the theorem to be generated: **Insert(Create, j) ≡ f*(Create, j)**
Normal form of **Insert(Create, j)**: **Insert(Create, j)**
Rules used for the normal form: None

Step (1) Invoke Synthesis Rule (1) on **Insert(Create, j)**
Insert(Create, j) ≡ Insert(Create, j)

Step (2) Expand Expression: **Insert(Create, j)**
Using Rule: (4)

Insert(Create, j) ≡ Rotate(Insert(Create, j)

The right hand side of the last theorem generated in the above series has the form of **f*(Create, j)**, and hence can be used to generate a set of candidate compositions. A candidate composition is determined from three expressions:

(1) the left hand side of the target implementation, say $F(v_1, \ldots, v_n)$

(2) the right hand side of the theorem generated, say α, and

(3) the right hand side of the template equation, say $f^*(g_1, \ldots, g_n)$.

It is obtained by replacing zero or more occurrences of g_i, for every $1 \leq i \leq n$, in α by a variable v_j, $1 \leq j \leq n$. The replacement of g_i by v_j is made so that type consistency is preserved.

For the current example, the left hand side of the target implementation is **ENQUEUE(d, k)** ::= ?; the right hand side of the theorem generated is **Rotate(Insert(Create, j)**; the right hand side of the template equation is **f*(Create, j)**. So, there are two candidates for **f*(d, k)**: (1) **Rotate(Insert(d, k))** and (2) **Rotate(Insert(Create, k)**.

The second candidate does not satisfy equation (2). The equation obtained by replacing **f*** in the equation by the candidate is **Insert(Rotate(Insert(Create, j)), i) ≡ Rotate(Insert(Create, j))**. This is not a theorem of **Circ_List** because (for every i and j) both the sides of the equation remain simplified, but will not be identical. (This can be checked by **Is-an-inductive-theorem-of.**)

Let us consider the first candidate. The equation obtained by substituting it for **f*** in equation (2) is **Rotate(Insert(Insert(c, i), j)) ≡ Insert(Rotate(Insert(c, j)), i)**, and this is a theorem of **Circ_List**. (The left hand side of the equation reduces to the right hand side by the rewrite rule (5).) Hence **Rotate(Insert(d, k))** satisfies equation (2). The second candidate does not satisfy equation (2). Hence the target implementation is:

ENQUEUE(d, k) ::= Rotate(Insert(d, k))

6.3 An Illustration of a Complete Synthesis

In the following, we illustrate the complete synthesis, i.e., an illustration of both the stages, of two examples. The first one derives a target implementation for the operation **Append** of **Queue_Int** using the association specification that specifies the **Circ_List** representation. The second example derives a target implementation for the **Front** using the association specification that specifies the **<Array_Int X Integer X Integer>** representation (see chapter 5).

Illustration 1

Stage 1:

Partial Preliminary Implementation of Append at Hand

APPEND(*c*, Create) → ?rhs$_1$
APPEND(*c*, Insert(*d*,*i*)) → ?rhs$_2$

Relevant Rewrite Rules of the Perturbed World

(10) Append(q, Nullq) → q
(14) ℋ(Create) → Nullq
(20) ℋ(ENQUEUE(c, i)) → Enqueue(ℋ(c), ℋ(i))}
(22) ℋ(APPEND(c, d)) → Append(ℋ(c), ℋ(d))

Derivation of the first rewrite rule

Form of the theorem to be generated: **ℋ(APPEND(*c*, Create)) ≡ ℋ(?rhs$_1$)**
Normal form of **ℋ(APPEND(*c*, Create))**: **ℋ(*c*)**
Rules used for the normal form: (22), (14), (10)

Step (1) Invoke Synthesis Rule (1) on **ℋ(APPEND(*c*, Create))**

$\mathcal{H}$(APPEND(*c*, Create)) ≡ $\mathcal{H}$(*c*)

The above theorem is such that APPEND(*c*, Create) ≻ *c*. Therefore the desired rewrite rule is:

APPEND(*c*, Create) → *c*

Derivation of the second rewrite rule

Form of the theorem to be generated: $\mathcal{H}$(APPEND(*c*, Insert(Create, *i*))) ≡ $\mathcal{H}$(?rhs$_2$)
Normal form of $\mathcal{H}$(APPEND(*c*, Insert(Create, *i*))): Enqueue($\mathcal{H}$(*c*), $\mathcal{H}$(*i*))
Rules used for the normal form:

Step (1) Invoke Synthesis Rule (1) on $\mathcal{H}$(APPEND(*c*, Insert(Create, *i*)))
$\mathcal{H}$(APPEND(*c*, Insert(Create, *i*))) ≡ Enqueue($\mathcal{H}$(*c*), $\mathcal{H}$(*i*))

Step (2) Expand Expression: Enqueue($\mathcal{H}$(*c*), $\mathcal{H}$(*i*))
Using Rule: (10)

$\mathcal{H}$(APPEND(*c*, Insert(Create, *i*))) ≡ Append(Enqueue($\mathcal{H}$(*c*), $\mathcal{H}$(*i*)), Nullq)

Step (3) Expand Expression: Nullq
Using Rule: (14)

$\mathcal{H}$(APPEND(*c*, Insert(Create, *i*))) ≡ Append(Enqueue($\mathcal{H}$(*c*), $\mathcal{H}$(*i*)), $\mathcal{H}$(Create))

Step (4) Expand Expression: Enqueue($\mathcal{H}$(*c*), $\mathcal{H}$(*i*))
Using Rule: (20)

$\mathcal{H}$(APPEND(*c*, Insert(Create, *i*))) ≡ Append($\mathcal{H}$(ENQUEUE(*c*, *i*)), $\mathcal{H}$(Create))

Step (5) Expand Expression: Append($\mathcal{H}$(ENQUEUE(*c*, *i*)), $\mathcal{H}$(Create))
Using Rule: (22)

$\mathcal{H}$(APPEND(*c*, Insert(Create, *i*))) ≡ $\mathcal{H}$(APPEND(ENQUEUE(*c*, *i*), Create))

Step (6) Generalize the theorem in step (5) by replacing the constant
Create by the variable d to obtain the following equation:
$\mathfrak{H}$(APPEND(*c*, Insert(*d*,*i*))) ≡ $\mathfrak{H}$(APPEND(ENQUEUE(*c*, *i*), *d*))

Apply Is-an-inductive theorem-of on the above equation.
This yields True *confirming that the equation is a theorem.*

Hence the desired rule (obtained by dropping $\mathfrak{H}$ on both sides) is:
APPEND(*c*, Insert(*d*,*i*)) → APPEND(ENQUEUE(*c*, *i*), *d*)

Stage 2:

Preliminary Implementation at Hand

APPEND(*c*, Create) → *c*
APPEND(*c*, Insert(*d*,*i*)) → APPEND(ENQUEUE(*c*, *i*), *d*)

Desired Form of Target Implementation

APPEND(v_1, v_2) ::= ??

Relevant Rules of Circ_list

(10) Join(c, Create) → c
(11) Join(c, Insert(d, i)) → Insert(Join(c, d), i)

Template Equation Chosen: *c* ≡ APPEND(*c*, Create)
Form of the theorem to be generated: *c* ≡ f*(*c*, Create)
Normal form of *c*: *c*
Rules used for the normal form: None

Step (1) Invoke Synthesis Rule (1) on *c*
c ≡ *c*

Step (2) Expand Expression: *c*
Using Rule: (10)

---

c ≡ Join(*c*, Create)

Step (3) Find a suitable candidate composition.

The right hand side of the above theorem has the form of $\mathbf{f^*}(c, \mathbf{Create})$. So, find a suitable candidate composition. There are two possibilities: (1) $\mathbf{Join}(v_1, v_2)$, and (2) $\mathbf{Join}(v_2, v_1)\}$. The second candidate satisfies the second rule of the preliminary implementation, but the first does not. So, a possible target implementation is:

$$\mathbf{APPEND}(v_1, v_2) ::= \mathbf{Join}(v_2, v_1)$$

Illustration 2

Stage 1:

Partial Preliminary Implementation of Append

FRONT(<v, i, i>) → ?rhs$_3$
FRONT(<Assign(v, e, i), i, i+1>) → ?rhs$_4$
FRONT(<Assign(Assign(v,e_1, j), e_2, j+1), i, j+2>) → ?rhs$_5$

Relevant Rewrite Rules of the Perturbed World

(1) $\mathcal{H}$(<v, i, i>) → Nullq
(2) $\mathcal{H}$(<Assign(v, e, j), i, j+1>) → if_then_else(i = j+1, Nullq, Enqueue($\mathcal{H}$(<v, i, j>), $\mathcal{H}$(e)))

(3) $\mathcal{H}$(FRONT(x)) → Front($\mathcal{H}$(x))
(4) $\mathcal{H}$(ERROR) → Error
(5) $\mathcal{H}$(if_then_else(b, v_1, v_2)) → if_then_else(b, $\mathcal{H}$(v_1), $\mathcal{H}$(v_2))

Derivation of the first rewrite rule

Form of the theorem to be generated: $\mathcal{H}$(FRONT(<v, i, i>)) ≡ $\mathcal{H}$(?rhs$_1$)
$\mathcal{H}$(FRONT(<v, i, i>))↓: Error
Rules used for simplification:

Step (1) Invoke Synthesis Rule (1) on $\mathcal{H}$(FRONT(<v, i, i>))
$\mathcal{H}$(FRONT(<v, i, i>)) ≡ Error

Step (2) Expand Expression: Error
Using Rule: (4)

$\mathcal{H}$(FRONT(<v, i, i>)) ≡ $\mathcal{H}$(ERROR)

FRONT(<v, i, i>) → ERROR

Derivation of the second rewrite rule

Form of the theorem to be generated: $\mathcal{H}$(FRONT(<Assign(v, e, i), i, i+1>)) ≡ $\mathcal{H}$(?rhs$_2$)
$\mathcal{H}$(FRONT(<Assign(v, e, i), i, i+1>))↓: $\mathcal{H}$(e)
Rules used for simplification:

Step (1) Invoke Synthesis Rule (1) on $\mathfrak{H}$(FRONT(<Assign(v, e, i), i, i+1>))

$\mathfrak{H}$(FRONT(<Assign(v, e, i), i, i+1>)) ≡ $\mathfrak{H}$(e)

FRONT(<Assign(v, e, i), i, i+1>) → e

Derivation of the third rewrite rule

Form of the theorem to be generated: $\mathfrak{H}$(FRONT(<Assign(Assign(v,e_1, j), e_2, j+1), i, j+2>)) ≡ $\mathfrak{H}$(?rhs$_3$)

$\mathfrak{H}$(FRONT(<Assign(Assign(v,e_1, j), e_2; j+1), i, j+2>))↓:

if_then_else(i = j+2, Error, if_then_else(i = j+1, $\mathfrak{H}$(e_2),

Front(Enqueue($\mathfrak{H}$(<v, i, j>), e_1))))

Rules used for simplification:

Step (1) Invoke Synthesis Rule (1)

$\mathfrak{H}$(FRONT(<Assign(Assign(v,e_1, j), e_2, j+1), i, j+2>)) ≡

if_then_else(i = j+2, Error, if_then_else(i = j+1, $\mathfrak{H}$(e_2),

Front(Enqueue($\mathfrak{H}$(<v, i, j>), e_1))))

Step (2) Expand Expression: Front(Enqueue($\mathfrak{H}$(<v, i, j>), e_1))

Using Rule: (2), Protocol 3

TW Update:

i = j+2 → **False**

i = j+1 → **False**

$\mathfrak{H}$(FRONT(<Assign(Assign(v,e_1, j), e_2, j+1), i, j+2>)) ≡

if_then_else(i = j+2, Error, if_then_else(i = j+1, $\mathfrak{H}$(e_2),

Front($\mathfrak{H}$(<Assign(v, e_1, j), i, j+1>))))

Step (3) Expand Expression: $\mathfrak{H}$(<Assign(v, e_1, j), i, j+1>)

Using Rule: (3)

$\mathfrak{H}$(FRONT(<Assign(Assign(v,e_1, j), e_2, j+1), i, j+2>)) ≡

if_then_else(i = j+2, Error, if_then_else(i = j+1, $\mathfrak{H}$(e_2),

$\mathfrak{H}$(FRONT(<Assign(v, e_1, j), i, j+1>))))

Step (4) Expand Expression: Error

Using Rule: (4)

$$\mathcal{H}(\text{FRONT}(\langle \text{Assign}(\text{Assign}(v,e_1,j), e_2, j+1), i, j+2\rangle)) \equiv$$
$$\text{if\_then\_else}(i = j+2, \mathcal{H}(\text{ERROR}), \text{if\_then\_else}(i = j+1, \mathcal{H}(e_2),$$
$$\mathcal{H}(\text{FRONT}(\langle \text{Assign}(v, e_1, j), i, j+1\rangle))))$$

Step (5) Expand Expression: $\text{if\_then\_else}(i = j+2, \mathcal{H}(\text{ERROR}), \text{if\_then\_else}(i = j+1, \mathcal{H}(e_2),$
$\mathcal{H}(\text{FRONT}(\langle \text{Assign}(v, e_1, j), i, j+1\rangle))))$

Using Rule: (5)

$$\mathcal{H}(\text{FRONT}(\langle \text{Assign}(\text{Assign}(v,e_1,j), e_2, j+1), i, j+2\rangle)) \equiv$$
$$\mathcal{H}(\text{if\_then\_else}(i = j+2, \text{ERROR}, \text{if\_then\_else}(i = j+1, e_2,$$
$$\text{FRONT}(\langle \text{Assign}(v, e_1, j), i, j+1\rangle))))$$

$$\text{FRONT}(\langle \text{Assign}(\text{Assign}(v,e_1,j), e_2, j+1), i, j+2\rangle) \rightarrow$$
$$\text{if\_then\_else}(i = j+2, \text{ERROR}, \text{if\_then\_else}(i = j+1, e_2,$$
$$\text{FRONT}(\langle \text{Assign}(v, e_1, j), i, j+1\rangle)))$$

Stage 2:

Preliminary Implementation at Hand

$$\text{FRONT}(\langle v, i, \Omega\rangle) \rightarrow \text{ERROR}$$
$$\text{FRONT}(\langle \text{Assign}(v, e, i), i, i+1\rangle) \rightarrow e$$
$$\text{FRONT}(\langle \text{Assign}(\text{Assign}(v,e_1,j), e_2, j+1), i, j+2\rangle) \rightarrow \text{if } i = j+2 \text{ then ERROR}$$
$$\text{else if } i = j+1 \text{ then } e_2$$
$$\text{else FRONT}(\langle \text{Assign}(v, e_1, j), i, j+1\rangle)$$

Let FRONT(<arr, pnt1, pnt2>) be the left hand side of the target implementation. We use a slightly different method than the one normally used for deriving the target implementation for **Front**. We use combination of the recursion preserving method and the recursion eliminating method. First, a composition that satisfies the first rewrite rule is determined separately; it is easy to see that this can be **ERROR**. Then, a composition that satisfies the second and the third rewrite rules is determined. The two compositions are then combined with the help of a boolean inverting expression to arrive at the target implementation. Note that the boolean inverting expression that characterizes the argument structure corresponding to the first rewrite rule is **pnt1 = pnt2**. Therefore, the desired form of the target implementation is as below. The expression that takes the place of the else clause is to be

determined so that the second and the third rewrite rules are satisfied.

Desired Form of the Target Implementation

FRONT(<arr, pnt1, pnt2>) ::= if pnt1 = pnt2 then ERROR
else ??

Relevant Rewrite Rules of Array_Int and Array_Int X Integer XInteger

The first two rules specify the **Read** operation of **Array_Int** that reads an element of an array. The third rewrite rule specifies the operation of a triple that selects the first component.

(1) Read(Nullarray, i) → ERROR
(2) Read(Assign(v, e, j), i) → if i = j then e
else Read(v, i)

(3) First(<v, k, l>) → v

Template equation chosen: $e \equiv$ **FRONT(<Assign(**v, e, i**),** $i, i+1$**>)**
Form of the theorem to be generated: $e \equiv$ **f*(<Assign(**v, e, i**),** $i, i+1$**>)**
Normal form of e: e
Rules used for simplification: None

Step (1) Invoke Synthesis Rule (1) on e

$e \equiv e$

Step (2) Expand Expression: e
Using Rule: (2), Protocol 2

$e \equiv$ **Read(Assign(v,** e**, i), i)**

Step (3) Expand Expression: **Assign(v,** e**, i)**
Using Rule: (3)

$e \equiv$ **Read(First(<Assign(v,** e**, i), k, l>), i)**

Step (4) Replace variables in the theorem by appropriate terminals:
v $\mapsto v$, **i** $\mapsto i$, **k** $\mapsto i$, **l** $\mapsto i+1$

$e \equiv$ **Read(First(<Assign(**v, e, i**),** i, $i+1$**>),** i**)**

The right hand side of the last theorem generated has the form of **f*(<Assign(**v, e, i**),** i, $i+1$**>)**. It determines the candidate composition **Read(First(<arr, pnt1, pnt2>), pnt1)**, which can be simplified to **Read(arr, pnt1)**. This composition is such that when it is takes the place of ?? in the partial target implementation shown above, the whole expression satisfies the third rewrite rule in the preliminary implementation. Hence, the a possible target implementation for **FRONT** is:

FRONT(<arr, pnt1, pnt2>) ::= if pnt1 = pnt2 then ERROR
else Read(arr, pnt1)

---

7. Conclusions and Future Research

Algebraic specifications for data types have been extensively used to prove properties of data types and to establish the correctness of implementations of data types. In this thesis we have investigated the task of automatically synthesizing implementations for abstract data types starting from their algebraic specifications. In this chapter we summarize the major contributions of the thesis, describe the important conclusions the research has lead us to, and provide directions for further research.

One of the main decisions that we were confronted with at the start of the research was choosing and characterizing the inputs to the synthesis procedure. It is not surprising to expect as inputs the specification of the implemented type, and the specifications of all the implementing types. The novelty of our method lies in the use of two other inputs: the homomorphism information and the termination ordering. The advantages of having them as inputs became more evident as the research progressed.

The homomorphism information makes the problem more tractable by restricting the space to be searched in finding an implementation because it imposes additional constraints on the synthesis equations (see chapter 4). It is informative in this respect to compare our method with that of Okrent's [40]. The method developed in [40] can also be reformulated as a theorem generation activity within our framework. His method, however, is less general and less efficient than ours because he does not use the homomorphism information. In order to compensate for the lack of this information he is forced to severely restrict the form of the specifications.

The termination ordering is not essential but is useful for automating the synthesis procedure. The basic method of manipulation used by the synthesis procedure is expansion (see section 4.4.1 and 4.5). Expansion, unlike reduction, is not uniformly terminating -- even when the specifications are convergent (see section 3.3). This makes the synthesis procedure potentially nonterminating. The termination ordering circumvents this problem. It also ensures the termination of the implementation derived. The synthesis method used by Darlington [7] does not explicitly indicate the use of any termination ordering. This is one of the reasons that the issue of termination (that of the synthesis procedure, or that of

implementation derived) is not addressed in [7].

An important contribution of the thesis is the development of a formal basis for the method used by the synthesis procedure. The development is influenced significantly by the techniques used for verifying the correctness of implementations of algebraically specified data types. The synthesis method has two distinguishing features. The first is that it is based on the general principle of reversing the techniques of program verification. The second is the decomposition of the procedure into two stages.

The reverse program verification principle lead us to view the synthesis problem (see chapter 4) as one of generating a set of theorems that satisfy the synthesis conditions. The synthesis conditions characterize the situations in which a set of theorems of the input specifications is guaranteed to yield a correct implementation. The synthesis rules provide a means of generating theorems from a specification. This approach to synthesis has two advantages. Firstly, it makes the formal justification of the correctness of the synthesis method simple because the synthesis conditions are based on a criterion of correctness for data types. Secondly, it allows us to build on the research in the area of program verification - past as well as future. This naturally suggests an area in which to pursue future research. It concerns extending the theory in which the synthesis procedure operates. Currently it operates in the part of inductive theory of the specification that is decided by the Musser/KB method (see chapter 4) of proving equational and inductive properties of rewriting systems. This extension would involve developing new synthesis rules, and new ways of using the synthesis rules for generating theorems. One might, for example, look into ways of assimilating the proof techniques used by various verifiers [5, 27] into our framework.

Another advantage of decomposing the procedure into two stages is that it makes the procedure more modular. It isolates the part that is dependent on the target language. So modifications to the target language can be made without drastically affecting the synthesis procedure. A possible extension to the thesis that could be considered is to incorporate more equivalence preserving transformations into the second stage. The transformations can be either of an efficiency improving nature, or language developing nature such as applicative to imperative transformations.

In addition to characterizing the inputs, an important contribution of the thesis is

the characterization of the generality of the synthesis method. The thesis formally characterizes (see chapter 2 and section 3.3) the restrictions on the inputs, and the conditions under which it succeeds in finding an implementation. This was possible primarily as a result of the development of the formal basis for the synthesis method.

Finally, but most importantly, let us address the question that any work on program synthesis has to confront: How far does the work go towards making the programmers task superfluous ? The practical utility of a work in program synthesis can be determined by evaluating the following aspects of the synthesis procedure: Efficiency of the synthesis method, efficiency of the implementations derived, and the ease of writing specifications.

The main source of inefficiency in the synthesis procedure stems from the non-uniquely terminating nature of expansion. This forces (as shown in section 4.5) the procedure to keep track of all possible expansion paths. The implementation of the procedure given in section 4.5 uses only the most obvious ways in which unproductive paths can be pruned. There are several avenues for further research in this area. One can investigate the use of various heuristic approaches for cutting down unproductive paths. Another possibility is to make better use of the invariant information available in the association specification. The procedure (see chapter 5) currently uses it just as one of the conditions to terminate the theorem generation activity. A better utilization of it would be to guide the theorem generation activity. For instance, it would be more useful if it were possible to deduce from the invariant specification certain structural properties of expressions that prevented them from satisfying the invariant. This could then be used to discontinue unproductive expansion paths during theorem generation. It is hard to extract this kind of information from an algebraic specification of $\mathfrak{I}$. It would be interesting to consider other means of specifying $\mathfrak{I}$ which can help this cause.

The synthesis procedure currently does not take into consideration the efficiency of its output in synthesizing an implementation. It derives the implementations that it is capable of deriving in increasing order of complexity (in terms of the number of reduction steps needed) of the proof of the implementations. The are two reasons for this. Firstly, we know of no good ways of specifying performance constraints for operations of data types within an algebraic framework. Secondly, it was beyond the scope of the current work to incorporate

automatic performance analysis of the implementations. There is some recent work being done in this area in [50] that is compatible with algebraic theory of data types. It would be interesting to investigate the interaction between our work and that of [50].

The main reason for choosing an equational language to express the inputs was because of the benefits it offers from a proof theoretical point of view. Equational specifications have generally been found hard to write. This is one of the factors that reduces the practical value of the procedure. It would be useful to extend the synthesis procedure to accept specifications in a language that is easier to write.

We believe that the goal of the research in program synthesis (and program verification) should not and cannot be to relieve the programmer completely of the burden of programming. Rather, it should be to help us gain a better insight into the science of programming. The insight gained can be utilized in several ways that are practically relevant, such as in the design of new programming languages, and in the development of program maintaining and program development [19, 49, 2, 3] systems. We believe that our work can be particularly useful in the latter area.

REFERENCES

1. Balzer, R., Automatic Programming, Technical Memo 1, ISI, Sept. 1972.

2. Balzer, R., Goldman, N., Wile, D., On the Transformational Implementation Approach to Programming, *Second International Software Engineering Conference*, Oct. 1976, SanFrancisco, CA, PP. 337.

3. Bauer, F.L., Partsch, H., Pepper, P., Wossner, H., Notes On The oject CIP: Outline of a Transformation System, Technische Universitat Munchen, TUM-INFO-7729, July 1977.

4. Bosyj, M., A Program for the Design of Procurement Systems, TR-160, Lab. for Computer Science, M.I.T., May 1976.

5. Boyer, R.S., Moore, J.S., A Computatinal Logic, ACM Monograph Series, Academic Press, Inc., 1979

6. On Automating the construction of Programs, AI-236, Stanford AI Project, Stanford, CA, (March 1974).

7. Burstall, R.M. and Darlington, J., A Transformation System for Developing Recursive Programs, *Journal of the Association for Computing Machinery*, Vol. 24, No. 1, Jan. 1977, pp. 44-67.

8. Darlington, J., A semantic approach to automatic program improvement, Ph.D. Th., Artif. Intl., U. of Edinburgh, Edinburgh, 1972.

9. Darlington, J., Application of program transformation to program synthesis, Proc. IRIA Symp. Proving and Improving Programs, Arc-et-Senans, France, pp.133-144.

10. Dershowitz, N., Orderings for term-rewriting Sytems, Department of Computer Science, U. of Illinois at Urbana-Champaign, Urbana, Illinois, UIUCDCS-R-79-987, Aug. 1979.

11. Dijkstra, E.W., Notes on Structured Programming, In *Structured Programming* (Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.), Academic Press, London and New York, 1972, PP. 1-81.

12. Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, N.J., 1976.

13. Floyd, R.W., Assigning Meanings to Programs, Proceedings of a *Symposium*

in Applied Mathematics, Vol. 19 as *Mathematical Aspects of Computer Science (Ed. Schwartz, J.T.), American Mathematical Society, Providence. R.I., 1967, PP. 1-32.*

14. Goguen, J.A., Thatcher, J.W., Wagner, E.G., "Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," *Current Trends in Programming Methodology*, Vol. IV, Data Structuring, (Ed. Yeh, R.T.), Prentice Hall (Automatic Computation Series), Englewood Cliffs, New Jersey, 1978.

15. Good, D., London, R., and Blesoe, W., An Interactive Program Verification System, *Proceedings of 1975 International Conference on Reliable Software*, April 1975.

16. Guttag, J.V., The specification and Application to Programming of Abstract Data Types. Ph. D. Thesis, University of Toronto, CSRG-59, 1975.

17. Guttag, J.V., Horowitz, E., and Musser, D.R., The Design of Data Type Specifications, ISI, USC, Marina del Rey, CA, ISI?RR-76-49, Nov. 1976.

18. Guttag, J.V., Horning, J.J., The Algebraic Specification of Abstract Data Types., *Acta Informatica* Vol. 10, No. 1, 1978, pp.27-52

19. Hewitt, C.E. and Smith, B., Towards a Programming Apprentice, *IEEE Transactions on Software Engg.*, Vol. SE-1, No.1, March 1975, PP. 26-45.

20. Hoare, C.A.R., Procedures and Parameters: An Axiomatic Approach, In *Symposium on Semantics of Algorithmic Languages* (ed. Engeler, E.) as *Lecture Notes in Mathematics*, No. 188, Springer Verlag, 1971, PP. 102-115.

21. Hoare, C.A.R., Proof of Correctness of Data Representations, *Acta Informatica* Vol. 1, No. 4, pp 271-281, 1972.

22. Huet, G., Hullot, J.M., Proofs by induction in equational theories with constructors, *in* 21st IEEE Symposium on Foundations of Computer Science (1980), 11, pp. 96-107.

23. Jouannaud J-P., Lescanne P., and Reinig F., Recursive Decomposition Ordering, Conference on Formal Description of Programming Concepts, Garmisch, (1982).

24. Kapur, D., Srivas, M.K., Expressiveness of the Operation Set of A Data Abstraction, Computation Structures Group Memo 179-1, Lab. for Computer Science, M.I.T., Cambridge, MA, June, 1979, Revised Nov., 1979.

25. Kapur, D., Towards a Theory for Abstract Data Types, TR-237, Lab. for Computer Science, M.I.T, Camb., MA 02139.

26. Kamin, S. An Informal Note on Extensions to recursive path Orderings INRIA. (Obtained Via Personal Communication with Pierre Lescanne.)

27. King, A Program Verifier, Ph.D. Thesis, Carnegie-Melon University, 1969.

28. Knuth, D.E., Bendix, P.B., Simple Word Problems in Universal Algebras, In *Computational Algebra (Ed. Leach, J.)*, Pergamon Press, 1970, pp. 263-297.

29. Lampson, B.W., Horning, J.J, London, R.L., Mitchell, J.G., Popek, G.L., Report on the Programming Language Euclid, *SIGPLAN Notices*, Vol. 12, No. 2, Feb. 1977. ,

30. Feather M.S., A System for Assisting Program Transformation, Transactions on Programming Languages and Systems, Vol. 4, No. 1, January 1982.

31. Liskov, B.H., A Design Methodology for Reliable Software Systems, Fall Joint Computer Conference, 1972.

32. Liskov, B.H., Snyder, A., Atkinson, R., Schaffert, C., Abstraction Mechanisms in CLU, *CACM* Vol. 20 No. 8, pp. 564-576, 1977.

33. Liskov, B.L., Snyder, A.S., Exception Handling In CLU. Computation Structures Group Memo 155-2, Lab. for Computer Science, M.I.T., Cambridge, MA, Dec., 1977, Revised March 1979. To appear in *IEEE Trans. on Software Engineering.*

34. Liskov, B.H., et. al., CLU Reference Manual, CSG Memo 160-1, Lab. for Computer Science, M.I.T, Oct. 1979.

35. Manna, Z., Ness, S., On the Termination of Markov Algorithms, Proceedings of *The Conference on Theoritical Computer Science*, Univ. of Waterloo, Waterloo, Ontario, pp.43-46.

36. Manna, Z. and Waldinger, R., A Deductive Approach to Program Synthesis, TOPLAS, Vol. 2, No. 1, Jan. 1980. PP. 90-121.

37. McCarthy, J., Recursive Functions of Symbolic Expressions and their Computation by Mahine, Part I, *CACM*, Vol. 3, No. 4, April 1960

38. Musser, D.R., On Proving Inductive Properties of Abstract Data Types, Conference Record of the *Seventh Annual ACM Symposium on Principles of*

Programming Languages, Las Vegas, Nevada, Jan. 28-30., pp.154-162.

39. Musser, D.R., Abstract Data Type Specification in the AFFIRM System, *Proceedings of the Specification of Reliable Software Conference*, Boston, April 3-5, 1979, pp.47-57.

40. Okrent, H.F., Synthesis of Data Structures from Their Algebraic Descriptions, Ph.D. Thesis, Dept. of Electrical Engg. and Computer Science, M.I.T., Cambridge, MA 02139, Feb. 1977.

41. Parnas, D.L., Information Distribution Aspects of Design Methodology, Technical Report, Dept. of Computer Science, Carnnègie-Melon University, 1977.

42. Clark, K.L., McCabe, F.G., The Control Facilities of IC-PROLOG, Published in Expert Systems in the Micro Electronic Age, Ed. Michie, D., Edinburgh University Press, 1979.

43. *Proceedings of ACM Conference on Language Design for Reliable Software, SIGPLAN Notices* 12, 3.

44. Robinson, J.A., A machine-oriented logic based on the resolution principle, *JACM* 12, 1 (Jan. 1965), pp.23-41

45. Rogers, H., Jr., *Theory of Recursive Functions and Effective Computability.* McGraw-Hill Series in Higher Mathematics, McGraw-Hill, Inc., 1967.

46. Rovner, P., Automatic Representation Selection for Associative Data Structures, Computer Science Department, University of Rochester, Tech.Rep. 10, Sept. 1976.

47. Rowe, L.A. and Tonge, F.M., Automating the Selection of Implementation Structures, IEEE Transactions on Software Engg., Vol. SE-4, No. 6, Nov. 1978

48. standish, T., Kibler, D., Neighbors, J., Improving and Refining Programs by Program Manipulation, *Proceedings of the 1976 ACM National Conference*, Houston, Texas, Oct. 1976, PP. 509-516.

49. Standish, T., Harriman, D., Kibler, D, and Neighbors, J., The Irvine Program Transformation Catalogue, Computer Science Department, U.C. Irvine, Irvine, CA Jan. 1976.

50. Subrahmanyam, P.A. An Automatic/Interactive Software Development System, Department of Computer Science, University Utah, Salt Lake City, Utah

84112

51. Tompa, F. and Gotlieb, C., Choosing a Storage Schema, University of Toronto, Computer Science Report No. 54, May 1973

52. Wulf, W., London, R.L., and Shaw, M., Abstraction and Verification in ALPHARD: Introduction to Language and Methodology, Carnegie-Mellon University Technical Report, also USC Information Sciences Institute Research Report, 1976.

Appendix I - Equations as Rewrite Rules

Automatic verification of data types that are specified equationally is often based on treating the equations in the specifications as rules for rewriting expressions that have certain patterns. The automation of our synthesis method also relies on such a treatment of the specifications. This appendix describes the basic concepts about rewrite rules, and some useful properties of sets of rewrite rules.

We assume a denumerable set (Ψ) of elements called *variables*, and a finite set Σ of function symbols. We define *expressions* and *constants* over Σ as follows. (The formal definition is similar to the informal one given back in sec.3.3.1.)

Expressions

An *expression* is either (1) a variable, or (2) a function symbol $\mathbf{f}$ followed by a sequence of $n \geq 0$ expressions $\mathbf{e_1}, \ldots, \mathbf{e_n}$. $\mathbf{f}$ is called the (main) function of this expression, and $\mathbf{e_1}, \ldots, \mathbf{e_n}$ are called the *arguments*. Such an expression is written $\mathbf{f(e_1, \ldots, e_n)}$. An expression with no arguments is written as $\mathbf{f}()$. We denote the set of expressions defined over Σ as $E(\Sigma)$.

We assume it is possible to test variables and function symbols for equality. Two expressions α and β are regarded as identically equal (written $\alpha \equiv \beta$) if and only if they are both the same variable or they have the same main function symbol and the same number of identically equal arguments, in the same order.

The *variable set* of an expression α is $\{\alpha\}$ if α is a variable, otherwise is the union of the variable sets of the arguments of α.

The *subexpressions* of an expression are (1) the entire expression, and (2) the subexpressions of the arguments (if any) of the expression. Expressions which are variables have no expressions other than themselves.

Constants

A *constant* is an expression that does not contain any variables. We denote the set of constants over Σ as $T(\Sigma)$. The *subconstants* of a constant are (1) the entire constant, and (2) the subconstants of the arguments (if any) of the constant.

Occurrences

An expression can be represented naturally as a tree structure: The main function symbol of the expression is the root of the tree; the arguments of the expression are the branches of the tree. This analogy can be used to devise a notation to identify unambiguously the subexpressions of an expression.

An *occurrence in* an expression is a sequence (possibly empty) of positive integers that denotes the path inside the tree corresponding to the expression that runs from the root of the tree to the root of the tree corresponding to one of the subexpressions. We denote the set of all occurrences in an expression $\mathbf{e}$ by $O(\mathbf{e})$. We use the following notation for denoting an occurrence: λ is the empty

occurrence, and if u is an occurrence and **i** is an integer, then **i**.u is the occurrence that has **i** at its head and u as its tail.

The subexpression of an expression **e** at the occurrence u, denoted by $\mathbf{e}/u$, is defined as follows:

If $u = \lambda$, then $\mathbf{e}/\lambda = \mathbf{e}$

If $u = \mathbf{i}.w$ $(1 \leq i \leq n)$, and $\mathbf{e} = \mathbf{f}(\mathbf{e}_1, \ldots, \mathbf{e}_n)$, then $\mathbf{e}/u = \mathbf{e}_i/w$

For example, suppose **e** = **Enqueue(Dequeue(Nullq( )), i)**. Then **e**/1 = **Dequeue(Nullq ())**, **e**/2 = **i**, **e**/1.1 = **Nullq( )**.

Suppose u is an occurrence of **e**. Then, we use the notation $\mathbf{e}[u \leftarrow \mathbf{e}']$ to denote the expression obtained by replacing in **e** the subexpression $\mathbf{e}/u$ by **e'**. For instance, suppose **e** is the same expression as in the example given above, and **e'** = **Nullq( )**, then **e**[1 ← **e'**] is **Enqueue(Nullq( ), i)**.

Substitutions

Let σ be a mapping from variables to expressions, such that $\sigma(\mathbf{v}) = \mathbf{v}$ for all but a finite number of variables **v**. Extend the domain of σ to the set of all expressions by defining $\sigma(\mathbf{f}(\mathbf{e}_1, \ldots, \mathbf{e}_n))$ to be $\mathbf{f}(\sigma(\mathbf{e}_1), \ldots, \sigma(\mathbf{e}_n))$. Such a mapping σ is called a *substitution* (of expressions for variables). The notation $\sigma = [\mathbf{v}_1 \mapsto \mathbf{e}_1, \ldots, \mathbf{v}_n \mapsto \mathbf{e}_n]$ will be used to denote the substitution σ such that $\sigma(\mathbf{v}_i) = \mathbf{e}_i$, for $1 \leq i \leq n$, and $\sigma(\mathbf{v}) = \mathbf{v}$.

We say that an expression β *has the form* of an expression α if there exists a substitution σ such that $\sigma(\alpha) = \beta$. For example, **Append(Nullq(), Enqueue(q, i))** has the form of **Append(q1, Enqueue(q2, i2))** by the substitution σ = **[q1 ↦ Nullq(), q2 ↦ q, i2 ↦ i]**. Notice that has the form of is not a symmetric relation.

Rewrite Rules

A *rewrite rule* is an ordered pair of expressions (L, R), such that the variable set of R is contained in the variable set of L. Usually (L, R) will be written L → R. A finite set of rewrite rules over a set of function symbols Σ is called a rewriting system over Σ. Let **R** be such a rewriting system.

An expression α is *reducible* with respect to **R** if there is a rule L → R in **R**, and an occurrence u of α such that α/u has the form of L. Let σ be a substitution such that $\sigma(L) = \alpha/u$, and $\beta = \alpha[u \leftarrow \sigma(R)]$. Then we say that α *directly reduces* to β (using **R**), and write it as $\alpha \rightarrow \beta$ (using **R**). Where the particular **R** in use is clear from the context, this will be written simply as $\alpha \rightarrow \beta$. If α is not reducible with respect to **R**, then we say α is *irreducible* with respect to **R**.

Let $\rightarrow^*$ be the smallest relation on pairs of expressions which is the reflexive, transitive closure of $\rightarrow$. Thus, $\alpha \rightarrow^* \beta$ if and only if there exist expressions $\alpha_0, \alpha_1, \ldots, \alpha_n$, where $n \geq 0$, such that $\alpha = \alpha_0$, $\alpha_i \rightarrow \alpha_{i+1}$ for $i = 0, \ldots, n\text{-}1$ and $\alpha_n = \beta$. We read $\alpha \rightarrow^* \beta$ as α *reduces to* β.

Suppose $\alpha \rightarrow^* \beta$, and β is irreducible. Then we say that α *simplifies* to β; β is called a *normal form* of α. We denote the normal form of **e** as $\mathbf{e}\downarrow$. A rewriting system **R** has the *unique termination property (UTP)* if the simplifies relation defined by **R** is a function; that is, every expression has at most one normal form in **R**.

A rewriting system **R** has the *finite termination property (FTP)* if there is no infinite sequence

$\alpha_0 \rightarrow \alpha_1 \rightarrow$ ... using $\mathbf{R}$.

A rewriting system $\mathbf{R}$ is *convergent* if it has FTP as well as UTP. In such a case, every expression in the system has exactly one normal form.

Appendix II - Checking Finite Termination

A general technique for proving termination of a rewriting system **R** with an alphabet Σ is to demonstrate that it is possible to define a well-founded partial ordering $\succ_R$ on $T(\Sigma)$ so that $t_1 \rightarrow t_2$ implies $t_1 \succ_R t_2$. A partial ordering is well-founded if there are no infinite descending sequences such as $t_1 \succ_R t_2 \succ_R \ldots$ for any constants. Hence, there cannot be any infinite sequence of rewrites using **R** also. The following theorem [Manna&Ness] provides a useful guideline to define a suitable partial ordering to prove FTP.

Theorem 3 A rewriting system **R** with an alphabet Σ satisfies FTP if there exists a well-founded partial ordering $\succ_R$ on $T(\Sigma)$ with the properties given below. We call a well-founded partial ordering that satisfies the following properties a *termination* ordering for the system **R** since the ordering can be used to show the termination of **R**.

(1) *Reduction:* For every rule $L \rightarrow R$ in **R**, and for every substitution σ of variables to constants, $\sigma(L) \succ_R \sigma(R)$.

(2) *Substitution:* $t \succ_R t'$ implies $f(...t...) \succ_R f(...t'...)$ for any constants t, t', $f(...t...)$, $f(...t'...)$ in $T(\Sigma)$.

The reduction condition asserts that applying any rule reduces the subterm to which the rule is applied in the well-founded ordering. The substitution condition guarantees that by reducing subterms the top-level constant is also reduced. Hence it follows that $t \rightarrow t'$ implies $t \succ_R t'$.

Fig. 20 gives a definition of a class of orderings called the *lexicographic recursive path* orderings ($\succ$). $\succ$ is parameterized with respect to an ordering ($>$) on the alphabet of a rewriting system. In addition to the substitution property mentioned in the above theorem, $\succ$ also contains the subterm relation: t_1 is a subterm of t_2 implies that $t_2 \succ t_1$. Such an ordering is usually referred to as a *simplification*

Fig. 20. The Lexicographic Recursive Path Ordering

Let $>$ be an ordering on an alphabet Σ. Then $\succ$ on $T(\Sigma)$ is defined as follows:

$s \succ t$ iff one of the following conditions is true

(1) $f > g \wedge s \succ t_i, 1 \leq i \leq n$

(2) $f = g \wedge \langle s_1, \ldots, s_n \rangle \succ\succ_{lex} \langle t_1, \ldots, t_n \rangle \wedge s \succ t_i, 1 \leq i \leq n$

(3) $(\exists s_i)\,[s_i = t \vee s_i \succ t]$

$\succ\succ_{lex}$ is a right to left lexicographic ordering based on $\succ$. It is defined as follows.

$\langle s_1, \ldots, s_n \rangle \succ\succ_{lex} \langle t_1, \ldots, t_n \rangle$ iff

$(\exists\, 1 \leq i \leq n)\,[s_i \succ t_i \wedge (\forall\, i < j \leq n)\,[s_j = t_j]]$

ordering [Dershowitz]. (A proof that $\succ$ is a simplification ordering can be found in [Kamin].) Dershowitz in [Dershowitz] has shown the following theorem:

Theorem 4 A lexicographic recursive path ordering ($\succ$) is well-founded if and only if the underlying alphabet ordering ($>$) is well-founded.

One can, in general, use any suitable well-founded alphabet ordering in conjunction with a lexicographic recursive path ordering to use it as a termination ordering for a rewriting system. Figures 21, and 22 give two alphabet orderings: The first can be used for an arbitrary data type specification, and the second for an arbitrary homomorphism specification. We refer to these orderings as the *standard* alphabet orderings for a data type specification, or a homomorphism specification, respectively. The orderings are based on a general method of structuring of the alphabets of a data type specification and a homomorphism specification. Assuming that there is no circularity in the defining_types relation on data types, it can be easily shown that the standard alphabet orderings are well-founded orderings.

A lexicographic recursive path ordering based on an alphabet ordering of Fig. 21 can serve as a termination ordering for the rewriting systems corresponding to **Queue_Int** and **Circ_List**. We leave it to the reader to convince for himself that $\succ$ satisfies the reduction property in each of the two cases;

Fig. 21. The Standard Alphabet Ordering for a Data Type Rewriting System

Notations

S is the rewriting system corresponding to TOI
Σ is the alphabet of S
Ω is the operation set of TOI
Ω_B is the set of generators of S
Ω_{NB} is the set of nongenerators of S
Σ_{Def} is the union of the alphabets of the rewriting systems of the defining types
(We assume that the alphabets are mutually exclusive.)

$>$ is a partial ordering on the symbols in Σ

Definitions

$\Sigma = \Omega_B \cup \Omega_{NB} \cup \Sigma_{Def}$
$\Omega = \Omega_B \cup \Omega_{NB}$

$>$ is defined as follows. It is assumed that a similarly defined ordering exists for each of the alphabets in Σ_{Def}. $>$ is assumed to contain each of these orderings.

$f > g$ iff one of the following conditions is true
(1) $f, g \in \Omega_B \wedge$ arity of $g = 0$, arity of $f > 0$
(2) $f \in \Omega_{NB} \wedge g \in \Omega_B$
(3) $f \in \Omega \wedge g \in \Sigma_{Def}$

Fig. 22. The Standard Alphabet Ordering for Homomorphism Specification

Notations

Σ_H is the alphabet of the homomorphism specification
$\succ_H$ is the standard alphabet ordering on Σ_H

Definition

$\mathbf{f} \succ_H \mathbf{g}$ if and only if one of the following conditions holds:

(1) **f** is the symbol $\mathfrak{H}$, and **g** is any other function symbol in Σ_H
(2) **f** is an auxiliary function symbol, and **g** is a generator function symbol

---

one needs to use the fact that $\succ$ contains the subterm relation in doing so. The ordering cannot, however, be used for **Array_Int** specification. The ordering can be used for a subset of the specification that is used in examples to illustrate the synthesis procedure. A lexicographic recursive path ordering based on the standard alphabet ordering of Fig. 22 can be used all of the sample homomorphism specifications given in the last chapter.

Lexicographic recursive path orderings are useful in defining termination ordering for a rewriting system that is built from two or more rewriting systems that have recursive path orderings already defined on them. Suppose $\succ_1$ and $\succ_2$ are two recursive path orderings defined with respect to the well-founded alphabet orderings $\succ_1$ and $\succ_2$, respectively. Suppose $\mathbf{R}_1$, and $\mathbf{R}_2$ are two systems for which $\succ_1$ and $\succ_2$ can serve as termination orderings. Then the recursive path ordering that is based on $\succ_1 \cup \succ_2$ can be used as a termination ordering for the system $\mathbf{R}_1 \cup \mathbf{R}_2$ provided $\succ_1 \cup \succ_2$ is well-founded. The standard alphabet ordering is such that the union of any two of them (defined on mutually exclusive alphabets) preserves the well-foundedness property. Hence it is useful in the context of combining systems of rewriting systems.

Appendix III - Proofs of Theorems

Theorem 6
Let S be a system that satisfies the principle of definition. Let $e_1 \equiv e_2$ be an equation so that e_1 and e_2 have at least one nongenerator function symbol in them. Then, $e_1 \equiv e_2$ is a theorem of S if $S \cup \{e_1 \rightarrow e_2\}$ also satisfies the principle of definition.

Proof The proof is by contradiction. Let us assume that $S \cup \{e_1 \rightarrow e_2\}$ satisfies the principle of definition, but $e_1 \equiv e_2$ is not a theorem of S.

If $e_1 \equiv e_2$ is not a theorem of S, then there exists a substitution σ that maps variables to generator constants so that $\sigma(e_1)$ and $\sigma(e_2)$ have distinct normal forms in S. Since S satisfies the principle of definition, $\sigma(e_1)$ and $\sigma(e_2)$ have unique normal forms that are generator constants; let the normal forms be t_1 and t_2, respectively ($t_1 \neq t_2$). Note that $\sigma(e_2)$ and sie2 are distinct from t_1 and t_2, respectively because the latter two are generator constants while the former two are not. Therefore, in the system $S \cup \{e_1 \rightarrow e_2\}$ we have the following situation:

$\sigma(e_1) \rightarrow \sigma(e_2) \rightarrow^+ t_2$, $\sigma(e_1) \rightarrow^+ t_1$, and $t_1 \neq t_2$.

Thus, $S \cup \{e_1 \rightarrow e_2\}$ violates the principle of definition. Contradiction.

Q.E.D.

Theorem 7
PW is a Perturbed World. Suppose

(1) e_1 is an expression so that for every substitution σ of variables to generator constants $\sigma(e_1)$ is reducible using PW, and

(2) $PW \cup \{e_1 \rightarrow e_2\}$ is convergent.

Then, $e_1 \equiv e_2$ is a theorm of PW.

Proof PW is convergent. Therefore, to show that $e_1 \equiv e_2$ is a theorem of PW, we have to show that for every substitution σ of the variables in e_1 and e_2 by generator terms of the appropriate type, $\sigma(e_1)$ and $\sigma(e_2)$ have the same normal forms.

The proof is by contradiction. Let us suppose that $PW \cup \{e_1 \rightarrow e_2\}$ is convergent, but $e_1 \equiv e_2$ is not a theorem of PW. This means, there exists a σ such that $t_1 = \sigma(e_1)\downarrow$ and $t_2 = \sigma(e_2)\downarrow$ are distinct. By the second premise of the theorem, therefore, we have the following situation in $PW \cup \{e_1 \rightarrow e_2\}$

$\sigma(e_1) \rightarrow \sigma(e_2) \rightarrow^+ t_2$

$\sigma(e_1) \rightarrow^+ t_1$ and $t_1 \neq t_2$.

Therefore, $PW \cup \{e_1 \rightarrow e_2\}$ is not convergent. Notice the need for the second premise. If we did not have this premise $\sigma(e_1)$ could be identical to t_1, in which case $PW \cup \{e_1 \rightarrow e_2\}$ is still convergent.

Q.E.D.

Theorem 8

A rewriting system **R** satisfies the principle of definition if it satisfies the following conditions:

(1) **R** is well-spanned.

(2) **R** has FTP.

(3) Every critical pair $\langle \alpha_1, \alpha_2 \rangle$ of **R** is such that $\alpha_1 \equiv \alpha_2$ is a theorem of **R**.

Proof The first two conditions ensure that every constant in **R** has at least one normal form, and that every normal form is a generator constant. The following argument shows that every constant has a unique normal form.

The proof is by contradiction. Suppose there exists a constant that has two distinct normal form. Then, according to the KB-theorem there exists a nonconvergent critical pair. This contradicts the third condition in the statement of the theorem. Contradiction.

Q.E.D.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MIT/LCS/TR-276

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): Automatic Synthesis of Implementations for Abstract Data Types from Algebraic Specifications.

5. TYPE OF REPORT & PERIOD COVERED: Technical Report June '82

6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TR-276

7. AUTHOR(s): Mandayam K. Srivas

8. CONTRACT OR GRANT NUMBER(s): DARPA N00014-75-C-0661

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
MIT Lab for Computer Science
545 Technology Square
Cambridge, Ma. 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS:
DARPA
1400 Wilson Boulevard
Arlington, Virginia 22217

12. REPORT DATE: June 1982

13. NUMBER OF PAGES: 161

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
Office of Naval Research
Department of the Navy
Information Systems Program
Arlington, Virginia 22217

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report):
Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

This document is approved for public release and sale, distribution unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

See back

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

See back

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Automatic Synthesis of Implementations for Abstract Data Types from Algebraic Specifications

Abstract

Algebraic specifications have been used extensively to prove properties of abstract data types and to establish the correctness of implementations of data types. This thesis explores an automatic method of synthesizing implementations for data types from their algebraic specifications.

The inputs to the synthesis procedure consist of a specification for the implemented type, a specification for each of the implementing types, and a formal description of the representation scheme to be used by the implementation. The output of the procedure consists of an implementation for each of the operations of the implemented type in a simple applicative language.

The inputs and the output of the synthesis procedure are precisely characterized. A formal basis for the method employed by the procedure is developed. The method is based on the principle of reversing the technique of proving the correctness of an implementation of a data type. The restrictions on the inputs, and the conditions under which the procedure synthesizes an implementation successfully are formally characterized.

Name and Title of Thesis Supervisor: John V. Guttag
Associate Professor of Computer Science and Engineering

Key Words and Phrases: Abstract Data Type, Algebraic Specification, Association Specification, Abstraction Function, Invariant, Preliminary Implementation, Target Implementation, Term Rewriting System, Principle of Definition, Reduction, Expansion.

---

1. This report is a minor revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science in December 1981 in partial fulfillment of the requirements for the degree of Doctor Philosophy.